APPLICATIONS OF THE CONVEXIFIED CONVOLUTIONAL NEURAL NETWORK

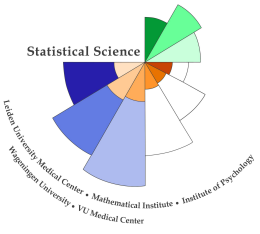
Experiments on simulated and real data

Maarten J.G. van Schaik (s1025325)

Thesis advisor: Dr. A.J. Schmidt-Hieber

MASTER THESIS

Specialization: Statistical Science



STATISTICAL SCIENCE FOR THE LIFE AND BEHAVIOURAL SCIENCES

Abstract

This thesis describes the model class of convexified convolutional neural networks (CCNNs), a type of deep learning model introduced by Zhang, Liang & Wainwright [1]. First, steps towards the convex relaxation are described, as well as all the steps required to implement the algorithm. To this end, the thesis describes the mathematical structure of the shallow networks, how the function class can be relaxed to the convex case, as well as the role of Reproducing Kernel Hilbert Spaces, the Nystrom method, and projected gradient descent on the nuclear norm ball. The main contribution of this work is the implementation and application to a new data set. The problems considered are a simulation study and an implementation on the classification problem of text data. The results of the CCNN implementation show that it can be successfully applied on text data through the use of vectorized word representations. Advantages and drawbacks compared to more mainstream approaches are discussed.

Contents

1 Introduction

This thesis will cover the subject of Convexified Convolutional Neural Networks (CCNNs), a type of deep learning model introduced by Zhang, Liang & Wainwright [1]. We will describe the motivation for these models, how they are modeled, and how they perform when applied to simulated and real-world text-based data sets.

The main contribution of this work is the implementation and application to a new data set. The problems considered are a simulation study and an implementation on the classification problem of text data. Because of the structural characteristics of the convolutional neural network (CNN), these models (and by extension the Convexified CNN) have been developed for classification tasks on image data. However, from a mathematical point of view, there is no specific reason why it should be only image-specific data which can be analyzed efficiently by these methods. In fact, using vector representation, text data can be represented as points in some high-dimensional space not unlike the representations given to image data. Therefore, after discussing what the CNN and CCNN models are and how they work, performance will be evaluated on text-based data sets and compared with more conventional methods.

In Section 2 we will cover the background for neural networks. We also discuss how they function from a conceptual point of view. Intuitively we can think of images as collections of features such as edges and angles, and the nodes as feature detectors. In Section 3 we discuss the networks from a mathematical point of view, where they are discussed as approximating functions stacked on each other in a hierarchical manner, resulting in a prediction by a function of the combined influence of all the parameters in the model. In particular, the variant of the convolutional neural network will be covered.

In Section 4 it is discussed how, given the function class of networks defined in the previous Section, a candidate network configuration is obtained by training algorithms. In particular, we note that the optimization algorithm is nonconvex and that no theoretical guarantees for the quality of the solution can be given.

In Section 5 the topic of convexification is introduced: how it is achieved through the use of the Reproducing Kernel Hilbert Space, and how Zhang, Liang & Wainwright [1] achieve this through particular kernel functions. The choice for kernel and activation functions will be discussed, but not too much expanded upon, since it falls outside the scope of this thesis.

Section 6 will introduce the topic of learning shallow (one hidden layer) CCNNs in practice. This part is based on [1], where the main steps of the algorithm are described. In this thesis, the algorithm will be described in some more detail, such as the use of the Nystrom method, the use of kernels, and projected gradient descent on the nuclear norm ball. This section will include a short discussion on (projected) gradient descent and how to solve constrained optimization problems using the method of Lagrange multipliers and KKT conditions.

The main contribution of this work is the implementation of the CCNN algorithm in Python, and the application to a new data set. Section 7 will show several applications of the CCNN and compare them to the more common nonconvex networks as well as SVM and logistic regression models. Several comparisons were made on simulated data, and in particular, emphasis will be placed on data of text format.

Finally, Section 8 will conclude the thesis and summarize the results.

2 General description of Artificial Neural Networks

An artificial neural network is based on the idea of a collection of highly interconnected processing units. Each processing unit can be seen as a simplified model of a biological neuron, and the artificial neurons are linked together, thus forming a network. Information can be sent through this network, with each neuron processing the information it receives using a simple function, and sending out a corresponding output. The connections between the neurons are adaptable in strength, with different configurations resulting in different network outputs. Then, the process of training involves finding the connections and their weights which are optimal for the assigned task. It turns out that the network as a whole, when trained, can sometimes perform very complicated predictions or calculations with impressive accuracy. Before the training, a network architecture must be chosen, and so far, many variants have been developed, such as convolutional neural networks [2], autoencoders [3, 4], and Deep Boltzmann Machines [5].

In recent years, network architectures have been developed which are capable of many tasks, including optical character recognition, text2speech / speech2text translation, robot arm control, face recognition, driving a car, and detecting credit card fraud.

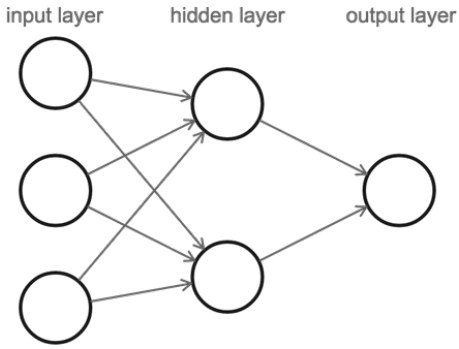


Figure 2: *(source: [6]) Example of a fully-connected shallow neural network of the form as in equation (3), with input layer, hidden layer, and a single output node. In this case, the dimensions are* $d_0 = 3$, $r = 2$, $d_2 = 1$. *The lines connecting the input to the hidden layer represent the weights* w, *the lines connecting the hidden to the output layer represent the weights* $\boldsymbol{\alpha}$.

3 The Shallow Neural Network

In this section, we give a detailed mathematical description of artificial neural networks. In what follows, whenever we refer to a 'shallow' network, we mean a network with a single hidden layer. In practical applications, networks with

multiple hidden layers are commonly used to learn deeper underlying representations of the data. In this thesis we will restrict ourselves to shallow networks. We define them as function classes for both the fully-connected and convolutional versions. In what follows, we use the notation $[n]$ to indicate the collection $\{1, 2, \ldots, n\}$, with n being a positive integer.

3.1 Shallow fully-connected neural networks

A shallow network is a function $f : \mathbb{R}^{d_0} \mapsto \mathbb{R}^{d_2}$ that maps an input vector $\mathbf{x} \in \mathbb{R}^{d_0}$ to an output vector $\mathbf{y} \in \mathbb{R}^{d_2}$ via a single hidden layer $h(\mathbf{x}) \in \mathbb{R}^r$. A simple model architecture of such a shallow network is a fully connected model, in which all the nodes in a layer are connected to all nodes in the previous (if any) and the next (if any) layer. An example of such a network is shown in Figure 2. After training, a *feedforward pass* through the network can be made to compute its output. For the fully-connected shallow network, this is performed as follows:

- First, the *hidden layer* activations are computed. Given some choice of activation function $\sigma : \mathbb{R} \to \mathbb{R}$ and a collection of weights $\{\mathbf{w}_j\}_{j=1}^r \in \mathbb{R}^{d_0}$ the following functions are computed:

$$h_j(\mathbf{x}) := \sigma(\mathbf{w}_j^{\mathrm{T}}\mathbf{x}) \tag{1}$$

  where each function h_j (for $j \in [r]$) is a node in the hidden layer of the network. Each of these hidden nodes has its own weight vector $\mathbf{w}_j$ representing the importance of the inputs $x_1, \ldots, x_{d_0}$ for that node.

- Then the activation of the *output layer* is computed. First we are given some choice of output activation function $\gamma : \mathbb{R} \to \mathbb{R}$ used by all output nodes. Then, coefficients $\boldsymbol{\alpha}_k \in \mathbb{R}^r$ define the contribution of the hidden nodes to the k-th output node, such that the total output of the k-th output node is calculated as:

$$f_k(\mathbf{x}) := \gamma\big(\sum_{j=1}^{r} \alpha_{kj} h_j(\mathbf{x})\big) \tag{2}$$

- In the last step, the complete output of the network is given by the concatenation over all output nodes:

$$f(\mathbf{x}) = \big(f_1(\mathbf{x}), \ldots, f_k(\mathbf{x}), \ldots, f_{d_2}(\mathbf{x})\big), \tag{3}$$

Generally, there is no restriction on the size that the parameters may take. However, for purposes later in this paper, it is convenient to introduce parameter constraints of (arbitrary, but fixed values) radii B_1 and B_2. The function class of these networks is the set/collection of all the functions of the following form:

$$\mathcal{N}(B_1, B_2) := \left\{ f^{\mathbf{w},\boldsymbol{\alpha}} : \max_{j\in[r]} \|\mathbf{w}_j\|_2 \leq B_1, \quad \max_{k\in[d_2]} \|\boldsymbol{\alpha}_k\|_2 \leq B_2 \right\}, \tag{4}$$

where $f^{\mathbf{w},\boldsymbol{\alpha}}$ indicates that the functions f in the function class depend on the collection of the parameter vectors $\mathbf{w} := \{\mathbf{w}_j \in \mathbb{R}^{d_0} : j \in [r]\}$ and $\boldsymbol{\alpha} := \{\boldsymbol{\alpha}_k \in \mathbb{R}^r : k \in [d_2]\}$.

3.2 Shallow Convolutional Neural Networks

The networks discussed up to this point form the basis for more elaborate extensions of the deep learning methods. One of these extensions is the convolutional neural network (CNN). CNNs have a use in a wide range of applications, but are especially useful for detecting spatial or temporal relationships between the input variables, where inputs near each other are more likely to be part of the same feature. This is particularly useful for (but not limited to) data representing images. An example is shown in Figure 3.

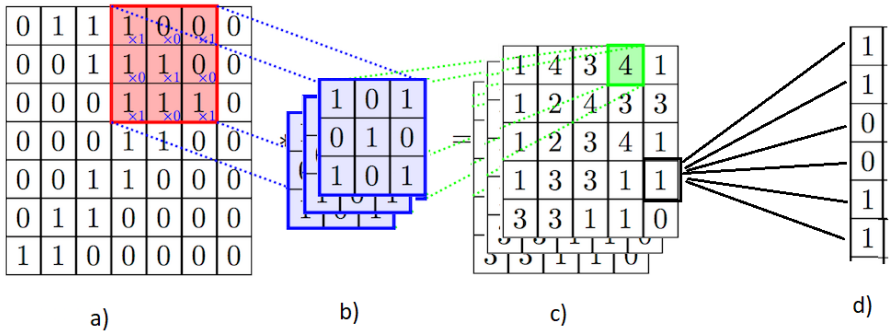


Figure 3: *(Adapted from [7]) Visualization of equation (7) applied to a* 7×7 *pixel image classification. a): Input vector* $\mathbf{x} \in \mathbb{R}^{d_0}$*. The red area is one patch vector* $\mathbf{z}(\mathbf{x}) \in \mathbb{R}^{d_1}$*. b): r weight vectors* $\mathbf{w} \in \mathbb{R}^{d_1}$*. c):* $h(\mathbf{z}(\mathbf{x}))$ *as green area. d): output vector* $\mathbf{y} \in \mathbb{R}^{d_2}$*, with coefficients* α_{kjp} *as black lines. Here,* $d_0 = 49$*,* $d_1 = 10$*,* $d_2 = 6$*,* $P = 25$*,* $r = 3$*.*

As a function, a CNN is a map from an input vector $\mathbf{x} \in \mathbb{R}^{d_0}$ to a d_2-dimensional output vector $\mathbf{y}$. This is still the same as previously described for the fully-connected network. However, as opposed to fully-connected networks, convolutional networks make use of parameter sharing and convolutional layers. Now, we will describe the feedforward pass through a shallow convolutional network:

- First, from each input $\mathbf{x}_i \in \mathbb{R}^{d_0}$ patches are constructed which represent local areas of the input. These patches $\mathbf{z}_p(\mathbf{x}_i) \in \mathbb{R}^{d_1}$ for $p \in [P]$ may (and in practice, typically will) consist of overlapping areas of $\mathbf{x}_i$.
- Then, the activation of each filter in the hidden layer is computed. Because of the convolutional property, each filter h_j is applied on all P patches $\mathbf{z}_p$ of the input $\mathbf{x}$, given their respective filter weights $\mathbf{w}_j \in \mathbb{R}^{d_1}$:

$$h_j(\mathbf{z}) := \sigma(w_{0j} + \mathbf{w}_j^T \mathbf{z}) \quad \text{for each patch} \quad \mathbf{z} \in \mathbb{R}^{d_1}. \tag{5}$$

  There are r of such filters in the hidden layer. Each filter is applied on each patch, and the weights for each particular filter are shared among all patches. This corresponds to the parameter sharing of a CNN.
- Then, given coefficients α_{kjp} which govern the contribution of the p-th patch to the k-th output via the j-th filter, the filters are used as contributions to an output function $f_k(\mathbf{x})$:

$$f_k(\mathbf{x}) = \sum_{j=1}^{r} \sum_{p=1}^{P} \alpha_{kjp} h_j(\mathbf{z}_p(\mathbf{x})) \tag{6}$$

- In the last step, the output of the network is the vector

$$f(\mathbf{x}) := (f_1(\mathbf{x}), \cdots, f_{d2}(\mathbf{x})). \tag{7}$$

  The parameters of this network are the filter vectors $\mathbf{w} := \{\mathbf{w}_j \in \mathbb{R}^{d1} : j \in [r]\}$ and the coefficient vectors $\boldsymbol{\alpha} := \{\boldsymbol{\alpha}_{k,j} \in \mathbb{R}^P, k \in [d_2], j \in [r]\}$.

Note that the weights $\mathbf{w}$ now have d_1 instead of d_0 dimensions, because the filters are applied on the patches $\mathbf{z}(\mathbf{x}) \in \mathbb{R}^{d_1}$ instead of the raw inputs $\mathbf{x} \in \mathbb{R}^{d_0}$. Like in the fully connected network, there is generally no restriction on the size that the parameters may take. Still, the introduction of radii B_1 and B_2 will help us later on in describing the convex relaxation. Furthermore, it is assumed that the patch vectors $\mathbf{z}_p(\mathbf{x}) \in \mathbb{R}^{d_1}$ are contained in the unit ℓ_2-ball. We consider the function class of CNNs as the set/collection of all the functions of the following form:

$$\mathcal{F}_{cnn}(B_1, B_2) := \Big\{ f \text{ of the form (7)} : \max_{j \in [r]} \|\mathbf{w}_j\|_2 \leq B_1, \max_{\substack{j \in [r] \\ k \in [d_2]}} \|\boldsymbol{\alpha}_{k,j}\|_2 \leq B_2 \Big\}. \tag{8}$$

Now we have defined our networks as the function classes as per equations (4) and (8). In the next Section we describe how a candidate network can be selected from the class.

3.3 The fully-connected versus convolutional architecture

Before continuing, we will note the main similarities and differences between the two networks described in the previous sections, and how clarify how these affects our choice of notation and terminlology. Both networks receive d_0-dimensional input and return d_2-dimensional output, but they differ in their hidden layer. In an attempt to keep notation consistent, we use r to indicate the number of functions in the hidden layers of both the fully-connected and the convolutional network. For a fully-connected layer, each function is a hidden node. For a convolutional layer, r indicates the number of filter functions. Only in the case of the convolutional network, P will be used to indicate the number of patches extracted from each input, and then d_1 will be used to indicate the dimension of the patches and filters. The fully-connected network does not use patches and parameter sharing and thus does not use the parameters P and d_1. Finally, we will indicate the parameters of the hidden layer $\mathbf{w}$ as "weights" and the parameters of the output layer $\boldsymbol{\alpha}$ as "coefficients". From a mathematical viewpoint there is no need to make this distinction, because they are treated identically for the purposes of training. However, the distinction is useful for ease of reading, and it also follows the choice of terminology used by Zhang et al. [1] on which parts of this text are based.

4 Finding the best network

In the previous section, we described the shallow neural networks, for both fully-connected and convolutional filters in the hidden layer. The collection of all possible networks are the function classes given by equations (4) and (8). In this section, we discuss the process of choosing a suitable network among the candidates of these classes, that this means solving a nonconvex optimization problem, and the consequences of this.

4.1 Loss functions and Empirical Risk Minimizer

The quality of a candidate solution is assessed by means of a loss function $\mathcal{L}(f)$: a function measuring the cost associated with using f. This cost is a measure of difference between the estimated and true values of the output of a training sample. The choice of loss function depends on the type of problem the network is trying to solve; for example, for categorical output cross-entropy can be used. In this thesis we concern ourself with a two-class classification problem where $y_i \in \{0, 1\}$. Then the task of learning the function $f_{\mathbf{A}} : X \mapsto Y$ concerns itself with the estimation of the class probabilities

$$\mathrm{P}(Y = y|\mathbf{A}, X = \mathbf{x}).$$

For two classes, this results in the probabilities like used in logistic or softmax regression:

$$\mathrm{P}_{\mathbf{A}}(\mathbf{x}_i) = \frac{1}{1 + \exp(-\mathbf{A}^\top \mathbf{x}_i)}, \tag{9}$$

and the loss function

$$\mathcal{L}(\mathbf{A}) = -\left[\sum_{i=1}^{n} y_i \log \mathrm{P}_{\mathbf{A}}(\mathbf{x}_i) + (1 - y_i) \log(1 - \mathrm{P}_{\mathbf{A}}(\mathbf{x}_i))\right]. \tag{10}$$

The goal of any fitting algorithm is then to find the parameters of the network such that the loss function on a training dataset (or a validation datset) is minimized. The question then arises whether it is possible to solve the problem of finding the best fitting function. The collection of optimal parameters are called the Empirical Risk Minimizer.

Definition 1 (Empirical Risk Minimizer)**.** Given observations X, a target Y, a function class $\mathcal{N} := \{f : X \to Y\}$ and a loss function $\mathcal{L}$, the candidate function f^* with the best fit is called the *Empirical Risk Minimizer* and is defined by

$$f^* = \inf_{f \in \mathcal{F}} \mathbb{E}_{X,Y}[\mathcal{L}(f(X); Y)].$$

4.2 Nonconvex optimization using Gradient Descent

Traditionally, finding the minimum loss is done using some kind of algorithm based on gradient descent. In these type of algorithms, the gradient of the loss function is taken with respect to the function parameters:

$$\nabla_{\mathbf{A}} \mathcal{L}(\mathbf{A}) = -\sum_{i=1}^{n} [\mathbf{x}_i \left(1\{y_i = k\} - \mathrm{P}(y_i = k|\mathbf{x}_i; \mathbf{A})\right)]. \tag{11}$$

In practical applications of neural networks, nearly all of the activation functions commonly used result in a nonconvex optimization problem. For fitting neural networks, gradient descent methods such as backpropagation must be used to train the network. Backpropagation will return a candidate solution $\hat{f}$, but this solution can only be considered the best fitting solution if we can prove that it is globally optimal, that is,

$$\mathcal{L}(f^*(X); Y) \leq \mathcal{L}(f(X); Y),$$

for every choice of f. When $\mathcal{L}$ and f are convex, standard theory of optimization tell us that global optimality of candidate f^* can be easily proven (if f^* is convex and the gradient $\nabla\mathcal{L}(f^*)$ vanishes to zero, then the parameters of f^* are the global solution). However, when f is nonconvex, and backpropagation has to be used for training, global optimality cannot be guaranteed.

4.3 Convex relaxation based on the nuclear norm

We have now reformulated the neural networks into a class of functions, which linearly depend on a matrix which is subject to a rank constraint. Given an affine subspace of matrices, we want to find the matrix adhering to the rank constraint and simultaneously minimizing a loss function (see Section 4.1). The problem with finding the optimal network subject to the rank constraint is that it is NP-hard. Therefore, no polynomial time algorithm is known to solve the problem. However, a nuclear norm constraint can be used to solve this problem indirectly. We first state the definition of the nuclear norm:

Definition 2 (Nuclear norm)**.** Let $\mathbf{A}$ be any $m \times n$-dimensional matrix and let the singular values of $\mathbf{A}$ be denoted by $\sigma(\mathbf{A}) \in \mathbb{R}^{\min(m,n)}$. Then the nuclear norm is defined by

$$\|\mathbf{A}\|_* = \mathrm{trace}(\sqrt{\mathbf{A}^T\mathbf{A}}) = \sum_{i=1}^{\min(m,n)} \sigma_i(\mathbf{A}).$$

In Section 5 we describe how the parameters of a shallow network can be collected in a parameter matrix. Due to the architecture of the network, this parameter matrix will be of a certain rank, and by the previously imposed parameter constraints B_1, B_2 it will have a constraint on the nuclear norm. We can use that matrix' nuclear norm as a convex relaxation of the matrix's rank constraint, as the following claim states:

Claim 1 (Nuclear norm as a convex relaxation)**.** *(By [8]) A standard convex relaxation of a rank constraint is based on the nuclear norm* $\|\mathbf{A}\|_*$ *corresponding to the sum of the singular values of* $\mathbf{A}$*.*

Thus, the nuclear norm constraint provides a proxy by which the rank constraint problem can be solved. Since the nuclear norm is a convex function of the network parameters, algorithms for solving convex problems can be used to

efficiently solve loss minimization problem. This approach was first introduced in [9], and in [8] it was shown that if certain properties hold, the minimum rank solution can be recovered by solving the minimization of the nuclear norm, which is a convex optimization problem. We will discuss such a method in Section 6. But first, the following Section describes the steps to achieve relaxation to a nuclear norm constraint for fully-connected and convolutional shallow networks.

5 Convexifying Shallow Neural Networks

In this section, we describe how the model class of shallow neural networks can be relaxed to a class of convexified shallow neural networks. We follow the method which Zhang et al. [1] used for convolutional neural networks, and apply them to the fully-connected neural network first. Also, we derive statements for rank and nuclear norm constraints. The first step for convex relaxation is to consider networks that use only the linear activation function, such that the function class of these networks implies a rank constraint of a parameter matrix. Then, we show that imposing bounds on the filter weights and output coefficients implies a bound on the nuclear norm of the parameter matrix.

Finally, we reconsider networks with certain nonlinear activation functions and show that by using Reproducing Kernels, these networks can also be convexified in the same manner. Which kernel and activation functions are suitable for this task was described by [1] and shortly repeated here in Section 5.2 for completeness.

5.1 Linear activation functions

5.1.1 Collecting the parameters in a matrix

The first step towards the convex relaxation of the class of network functions is to collect all parameters in a parameter matrix. According to Lemma 1 below, this can be achieved by restricting the network to linear activation functions.

Lemma 1 ((Based on [1]) In a shallow fully-connected neural network, linear activation functions allow for parameter collection in a matrix)**.** *Consider a shallow fully-connected network where the activation of each of the k output nodes takes the form (2) such that the output of the entire network takes the form (3). When using the linear activation functions $\sigma(t) = \gamma(t) = t$, the output of the k-th node can be written as*

$$f_k(\mathbf{x}) = \mathbf{x}^T \mathbf{a}_k \quad \text{for a vector } \mathbf{a}_k \in \mathbb{R}^{d_0} \tag{12}$$

such that the output of the whole network can be defined as

$$f^{\mathbf{A}}(\mathbf{x}) := (\mathbf{x}^T \mathbf{a}_1, \dots, \mathbf{x}^T \mathbf{a}_{d_2}) \tag{13}$$

In particular, $f^{\mathbf{A}}(\mathbf{x})$ depends linearly on the d_0-by-d_2-dimensional parameter matrix $\mathbf{A} \in \mathbb{R}^{d_0, d_2}$, which is the concatenation of the vectors $\mathbf{a}_k$, $k \in [d_2]$.

Proof. Each output of the network (2), with linear activation functions becomes $f_k(\mathbf{x}) = \sum_{j=1}^{r} \alpha_{kj} \langle \mathbf{x}, \mathbf{w}_j \rangle$. We can then see that

$$\begin{aligned} f_k(\mathbf{x}) &= \sum_{j=1}^{r} \alpha_{kj} \langle \mathbf{x}, \mathbf{w}_j \rangle \\ &= \mathbf{x}^{\mathrm{T}} \sum_{j=1}^{r} \alpha_{kj} \mathbf{w}_j \\ &= \mathbf{x}^{\mathrm{T}} \sum_{j=1}^{r} \mathbf{a}_{kj} \\ &= \mathbf{x}^{\mathrm{T}} \mathbf{a}_k \end{aligned}$$

such that the k-th output depends linearly on the d_0-dimensional parameter vector $\mathbf{a}_k \in \mathbb{R}^{d_0}$. Then, if we let $\mathbf{A} := (\mathbf{a}_1, \dots \mathbf{a}_{d2})$ be the concatenation over these parameter vectors across all d_2 output nodes, we see that the network as in equation (3) can be written in the form of (13). □

In the same manner that can be used to collect the parameters in a shallow fully-connected network, the convolutional neural network parameters can also be collected. Here, we show the steps of how this is done.

Lemma 2 (In a CNN, linear activation functions allow parameter collection in a matrix (as per [1]))**.** *Consider a CNN of the form (6 and 7), where the linear activation function $\sigma(t) = t$ is chosen and for each sample in the training set, the patches are collected in the matrix $\mathbf{Z}(\mathbf{x})$. Then the k-th output of the network can be written as*

$$f_k(\mathbf{x}) = tr(\mathbf{Z}(\mathbf{x})\mathbf{A}_k) \quad \text{for a matrix } \mathbf{A}_k \in \mathbb{R}^{d_1, P} \tag{14}$$

and the entire CNN output can be written as

$$f^{\mathbf{A}}(\mathbf{x}) := \big(tr(\mathbf{Z}(\mathbf{x})\mathbf{A}_1), \dots, tr(\mathbf{Z}(\mathbf{x})\mathbf{A}_{d_2})\big). \tag{15}$$

Proof. The proof for this follows by the same logic as shown in Lemma (1). This time, the input $\mathbf{x} \in \mathbb{R}^{d_0}$ is first replaced by the patches $\mathbf{z}_p(\mathbf{x})$ as described in Section 3.2. For each of the samples, we collect these patches in a P-by-d_1-dimensional matrix $\mathbf{Z}(\mathbf{x}_i) \in \mathbb{R}^{P, d_1}$. We also define the coefficient vectors $\boldsymbol{\alpha}_{kj} \in \mathbb{R}^P$ as the collected patch coefficients α_{kjp} for each of the k, j filter-output combinations. The parameters of the k-th output node can be collected in the d_1-by-P-dimensional matrix $\mathbf{A}_k = \sum_{j=1}^{r} \mathbf{w}_j \boldsymbol{\alpha}_{kj}^{\mathrm{T}}$. The p-th column of $\mathbf{A}_k$ is the d_1-dimensional vector $\sum_{j=1}^{r} \mathbf{w}_j \boldsymbol{\alpha}_{kjp} \in \mathbb{R}^{d_1}$. Since the p-th row of $\mathbf{Z}(\mathbf{x}_i)$ is the patch $\mathbf{z}_p(\mathbf{x})$, the diagonal elements of the matrix $\mathbf{Z}(\mathbf{x}_i)\mathbf{A}_k \in \mathbb{R}^{P,P}$ are $\mathbf{z}_p(\mathbf{x}_i)^{\mathrm{T}} \sum_{j=1}^{r} \mathbf{w}_j \alpha_{kjp}$, such that the trace over $\mathbf{Z}(\mathbf{x}_i)\mathbf{A}_k$ gives the desired output of the k-th node:

$$
\begin{aligned}
f_k(\mathbf{x}_i) &= \sum_{j=1}^{r} \sum_{p=1}^{P} \alpha_{kjp} \langle \mathbf{z}_p(\mathbf{x}_i) \mathbf{w}_j \rangle \\
&= \sum_{p=1}^{P} \mathbf{z}_p(\mathbf{x}_i)^{\mathrm{T}} \sum_{j=1}^{r} \mathbf{w}_j \alpha_{kjp} \\
&= \mathrm{tr}(\mathbf{Z}(\mathbf{x}_i) \mathbf{A}_k).
\end{aligned}
$$

Thus, each f_k depends linearly on the parameter matrix $\mathbf{A}_k$. Combining the matrices in the concatenation $\mathbf{A}$ results in the network parameter matrix $\mathbf{A} := (\mathbf{A}_1, \dots \mathbf{A}_{d2})$, which is a matrix of dimension d_1 by $d_2 \times P$.

□

5.1.2 The parameter matrix has a low rank bound

Now that we have collected the network's parameters in a matrix, we can describe the implied function of the network by the rank of said matrix. Now we show that there is a low rank bound on this matrix.

Lemma 3 (Low rank constraint in shallow NNs with linear activation functions (based on [1]))**.** *Let* $\mathbf{A}$ *be as in Lemma 1. Then rank*$(\mathbf{A}) \leq \min(d_0, d_2, r)$.

Proof. In Lemma 1, it was shown how the parameters of the shallow fully connected network can be collected in a single matrix $\mathbf{A}$. Another way to show this, is to gather the collection of filter weights $\{\mathbf{w}_j\}$ in a d_0-by-r-dimensional matrix $\mathbf{W} \in \mathbb{R}^{d_0, r}$ and to gather the output node coefficients $\{\boldsymbol{\alpha}_{kj}\}$ in a r-by-d_2-dimensional matrix $\mathbf{V} \in \mathbb{R}^{r, d_2}$. This way, the j-th column of $\mathbf{W}$ are the filter weights of the r-th hidden node $\mathbf{w}_j$, and the j-th row of $\mathbf{V}$ is the concatenation $(\boldsymbol{\alpha}_{1j}, \dots, \boldsymbol{\alpha}_{d_2 j})$. Then $\mathbf{A} = \mathbf{W}\mathbf{V}$. $\mathbf{W}$ is of dimension d_0 by r and of full rank. Similarly, $\mathbf{V}$ is of dimension r by d_2 and of full rank. Hence,

$$
\begin{aligned}
\mathrm{rank}(\mathbf{A}) &= \mathrm{rank}(\mathbf{W}\mathbf{V}) \\
&\leq \min(\mathrm{rank}(\mathbf{W}), \mathrm{rank}(\mathbf{V})) \\
&\leq \min(d_0, d_2, r).
\end{aligned}
$$

□

Lemma 4 (Low rank constraint in the shallow CNN (based on [1]).)**.** *When, in a shallow CNN using linear activation functions, the parameters have been collected in a matrix* $\mathbf{A}$ *as described in Lemma 2, this rank of this matrix is bounded: rank*$(\mathbf{A}) \leq \min(d_1, Pd_2, r)$.

Proof. As with the fully-connected network (Lemma 3), the parameter matrix $\mathbf{A}$ in the shallow CNN can also be decomposed in the matrices $\mathbf{W}$ and $\mathbf{V}$, where $\mathbf{W}$ is the collection of the filter weights $\{\mathbf{w}_j\}$, such that the j-th column of $\mathbf{W}$ is $\mathbf{w}_j \in \mathbb{R}^{d_1}$, and the j-th row of $\mathbf{V}$ is the concatenation of the d_2 P-dimensional vectors $\boldsymbol{\alpha}_{kj} \in \mathbb{R}^P$. $\mathbf{W}$ is of dimension d_1 by r and of full rank. Similarly, $\mathbf{V}$ is of dimension r by $d_2 P$ and of full rank. Hence,

$$\begin{aligned}\operatorname{rank}(\mathbf{A}) &= \operatorname{rank}(\mathbf{WV}) \\ &\leq \min(\operatorname{rank}(\mathbf{W}), \operatorname{rank}(\mathbf{V})) \\ &\leq \min(d_1, d_2 P, r).\end{aligned}$$

□

5.1.3 Nuclear norm bounds

As described in Section 4.3, the convexified CNN in [1] is motivated by the fact that the nuclear norm of the parameter matrix $\mathbf{A}$ is a convex relaxation of the rank constraint of $\mathbf{A}$. We will first show that the nuclear norm is indeed bounded.

Lemma 5 (Nuclear norm constraint (based on [1]))**.** *Take the class of shallow fully-connected networks, in both cases with the network parameters collected in the matrix* $\mathbf{A}$ *as shown in Lemma 1 and Lemma 2 by the use of linear activation functions. Then, in both these types of networks, if the network weights and coefficients are upper bounded, such that* $\|\mathbf{w}_j\|_2 \leq B_1$ *and* $\|\boldsymbol{\alpha}_{kj}\|_2 \leq B_2$ $\forall j \in [r]$, $\forall k \in [d_2]$*, then* $\mathbf{A}$ *must have a nuclear norm bounded as* $\|\mathbf{A}\|_* \leq d_2 r B_1 B_2$.

Proof.

$$\|\mathbf{A}\|_* := \operatorname{tr}((\mathbf{A}^{\mathrm{T}}\mathbf{A})^{1/2}) \tag{16}$$

$$= \operatorname{tr}\Big(\Big(\sum_{k=1}^{d_2} \mathbf{A}_k^{\mathrm{T}}\mathbf{A}_k\Big)^{1/2}\Big) \tag{17}$$

$$= \sum_{k=1}^{d_2} \operatorname{tr}((\mathbf{A}_k^{\mathrm{T}}\mathbf{A}_k)^{1/2}) \tag{18}$$

$$= \sum_{k=1}^{d_2} \|\mathbf{A}_k\|_* \tag{19}$$

In the above, steps (16) and (19) follow from the definition of the nuclear norm. Steps (17) and (18) follow from the fact that $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ is block matrix with as (i,j)-th block element the matrix $\mathbf{A}_i^{\mathrm{T}}\mathbf{A}_j$, $i,j \in [d_2]$. Because this matrix is symmetric and positive semidefinite, the sum of its singular values is equal to its trace. Because it is a block matrix, its trace is also equal to the sum of the traces of the matrices $\mathbf{A}_k^{\mathrm{T}}\mathbf{A}_k$, $k \in [d_2]$. Hence, the sum of its singular values (the nuclear norm) is equal to the sum of the nuclear norms of the matrices $\mathbf{A}_k$ that lie on its diagonal. These nuclear norms can be found as follows:

$$
\begin{aligned}
\|\mathbf{A}_k\|_* &= \|\sum_{j=1}^{r} \mathbf{A}_{kj}\|_* && (20)\\
&\leq \sum_{j=1}^{r} \|\mathbf{A}_{kj}\|_* && (21)\\
&= \sum_{j=1}^{r} \mathrm{tr}((\mathbf{A}_{kj}^{\mathrm{T}}\mathbf{A}_{kj})^{1/2}) = \sum_{j=1}^{r} \mathrm{tr}\Big(\sqrt{\mathbf{A}_{kj}^{\mathrm{T}}\mathbf{A}_{kj}}\Big) && (22)\\
&= \sum_{j=1}^{r} \mathrm{tr}\Big(\sqrt{(\mathbf{w}_j\boldsymbol{\alpha}_{kj}^{\mathrm{T}})^{\mathrm{T}}(\mathbf{w}_j\boldsymbol{\alpha}_{kj}^{\mathrm{T}})}\Big)\\
&= \sum_{j=1}^{r} \mathrm{tr}\Big(\sqrt{(\boldsymbol{\alpha}_{kj}\mathbf{w}_j^{\mathrm{T}})(\mathbf{w}_j\boldsymbol{\alpha}_{kj}^{\mathrm{T}})}\Big)\\
&= \sum_{j=1}^{r} \mathrm{tr}\Big(\sqrt{\boldsymbol{\alpha}_{kj}\|\mathbf{w}_j\|_2^2\boldsymbol{\alpha}_{kj}^{\mathrm{T}}}\Big)\\
&= \sum_{j=1}^{r} \|\mathbf{w}_j\|_2 \mathrm{tr}\Big(\sqrt{\boldsymbol{\alpha}_{kj}\boldsymbol{\alpha}_{kj}^{\mathrm{T}}}\Big)\\
&= \sum_{j=1}^{r} \|\mathbf{w}_j\|_2 \mathrm{tr}\Big(\sqrt{\boldsymbol{\alpha}_{kj}^{\mathrm{T}}\boldsymbol{\alpha}_{kj}}\Big) && (23)\\
&= \sum_{j=1}^{r} \|\mathbf{w}_j\|_2 \|\boldsymbol{\alpha}_{kj}\|_2 && (24)\\
&= rB_1B_2.
\end{aligned}
$$

In the above, steps (20) and (22) follow from the definition of the nuclear norm and $\mathbf{A}_k$. Step (21) is due to the sub-additive property of norms and step (23) from the property that $\|\mathbf{M}\|_* = \|\mathbf{M}^{\mathrm{T}}\|_*$ for any matrix $\mathbf{M}$. Combining all the above, we see that $\|\mathbf{A}\|_* \leq d_2 r B_1 B_2$, completing the proof. □

5.2 Nonlinear activation functions

In the previous section, we described how the class of neural networks can be relaxed to be convex when they are restricted to use the linear activation function. However, in the field of deep learning linear activation functions are typically not of interest. [1] describe how for certain nonlinear activation functions σ, and properly chosen kernel functions k, the class of CNN filters can be relaxed to a Reproducing Kernel Hilbert Space (RKHS), such that the problem can be of the same form as the linear activation case described in Section 5.1.1. To keep this thesis self-contained, we first give a brief overview of kernels and the RKHS. This overview is based on work by [10].

5.2.1 Kernel functions and the RKHS

Kernel functions are commonly used in many statistical learning applications for their ability to implicitly calculate similarities between data in a higher dimen-

sional space without exactly needing to know the representation in said high-dimensional space. Kernel methods are commonly used in applications such as the Support Vector Machine. Take any empirical data set $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m) \in \mathcal{X} \times \mathbb{R}$. Kernel functions k can be thought of as generalized dot products, and they are defined as

$$k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}, \quad (\mathbf{x}, \mathbf{x}') \mapsto k(\mathbf{x}, \mathbf{x}'). \tag{25}$$

The matrix of k being applied to all pairwise combinations of training data $(\mathbf{x}_i, \mathbf{x}_j)$, $i, j \in [m]$ are collected in the m-by-m-dimensional kernel matrix $\mathbf{K} := (k(\mathbf{x}_i, \mathbf{x}_j))_{ij}$. If this matrix is positive semidefinite, that is,:

$$\sum_{i=1}^{m} \sum_{j=1}^{m} c_i c_j \mathbf{K}_{ij} \geq 0 \quad \text{for all vectors } (\mathbf{c}_1, \ldots, \mathbf{c}_m) \in \mathbb{R}^m, \tag{26}$$

and where $\mathbf{K}_{ij} := k(\mathbf{x}_i, \mathbf{x}_j)$, then k is referred to as a positive semidefinite kernel function. Each kernel k is associated with a feature space using a function which maps from $\mathcal{X}$ into the space of functions mapping $\mathcal{X}$ into $\mathbb{R}$:

$$\phi : \mathcal{X} \to \mathbb{R}^{\mathcal{X}}, \quad \mathbf{x} \mapsto k(\cdot, \mathbf{x}). \tag{27}$$

The space of all functions that can be expressed as the linear combination of kernel functions is denoted as the Reproducing Kernel Hilbert Space $\mathcal{H}$. This means that any $f, g \in \mathcal{H}$ take the form of a linear combination:

$$f, g \in \mathcal{H} \Rightarrow f(\cdot) = \sum_i \alpha_i k(\cdot, \mathbf{t}_i)$$
$$g(\cdot) = \sum_i \beta_i k(\cdot, \mathbf{t}'_i),$$

(where the sums can be taken over countably many elements), and the inner product of the RKHS is defined by

$$\langle f, g \rangle_{\mathbb{H}} = \sum_{i,j} \alpha_i k(\mathbf{t}_i, \mathbf{t}'_j) \beta_j.$$

What gives the kernel its name of a reproducing kernel is that is has the property

$$\langle f, k(\cdot, \mathbf{x}) \rangle = f(\mathbf{x})$$

What this means in words is that *the inner product between some function* f *and some kernel function* k *which takes as its first argument* f *and as second argument some vector* $\mathbf{x}$ *returns as outcome a scalar which is equal to the function* f *evaluated at vector* $\mathbf{x}$. The fact that this operation returns $f(\mathbf{x})$ gives it its name of a *reproducing kernel*.

5.2.2 Representer theorem

The representer theorem by [10] (See Theorem 1 in Appendix A.1) states that optimization problems over a class of functions in an RKHS have solutions that

can be expressed as kernel expansions in terms of the training data. Furthermore, [10] describe how the representer theorem implies that for any training example $\mathbf{x}_j$ and any function $f \in \mathcal{H}$, application of f to $\mathbf{x}_j$ yields

$$f(\mathbf{x}_j) = \sum_{i=1}^{m} \alpha_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle. \tag{28}$$

The number of coefficients in functions of this form is determined by the sample size and thus finite. We now describe the relevance of this to the convexified network.

Lemma 6 (CCNN parametarization in terms of kernels)**.** *(By [1]). Consider a CNN of the form (15) and a training set of patches* $S := \{\mathbf{z}_p(\mathbf{x}_i) : p \in [P], i \in [n]\}$*. Using certain nonlinear activation functions* σ *and a positive semidefinite kernel function* $k : \mathbb{R}^{d_1} \times \mathbb{R}^{d_1} \to \mathbb{R}$*, then for any patch* $\mathbf{z}_p(\mathbf{x}_i) \in S$*, the filter values in Lemma (2) can be represented by*

$$h(\mathbf{z}_p(\mathbf{x}_i)) = \sum_{(i',p') \in [n] \times [P]} c_{i',p'} k\big(\mathbf{z}_p(\mathbf{x}_i), \mathbf{z}_{p'}(\mathbf{x}_{i'})\big), \tag{29}$$

for some coefficients $\{c_{i',p'}\}_{(i',p') \in [n] \times [P]}$*.*

[1, section 3.2 and Appendix B] describe the implications of Lemma 6. Because of the representer theorem, there exists some mapping ϕ such that $k\big(\mathbf{z}_p(\mathbf{x}_i), \mathbf{z}_{p'}(\mathbf{x}_{i'})\big) = \langle \phi(\mathbf{z}_p(\mathbf{x}_i)), \phi(\mathbf{z}_{p'}(\mathbf{x}_{i'})) \rangle$. If the kernel matrix $\mathbf{K} \in \mathbb{R}^{nP,nP}$ were to be approximated such that $\mathbf{K} \approx \mathbf{Q}\mathbf{Q}^{\mathrm{T}}$ where $\mathbf{Q} \in \mathbb{R}^{nP,m}$, and we subsequently set $\mathbf{q}_p(\mathbf{x}_i) \in \mathbb{R}^m$ as the m-dimensional feature vector of the training patch $\mathbf{z}_p(\mathbf{x}_i) \in \mathbb{R}^{d_1}$, then equation (29) can be rewritten as

$$h(\mathbf{z}_p(\mathbf{x}_i)) = \langle \mathbf{q}_p(\mathbf{x}_i), \bar{\mathbf{w}} \rangle \quad \text{where} \quad \bar{\mathbf{w}} := \sum_{(i',p') \in [n] \times [P]} c_{i',p'} \mathbf{q}_{p'}(\mathbf{x}_{i'}). \tag{30}$$

This means that from Lemma 6 we can see that in order to learn the filter h, it suffices to learn the m-dimensional vector $\bar{\mathbf{w}}$. This is done by replacing the patches $\mathbf{Z}(\mathbf{x}_i) \in \mathbb{R}^{P,d_1}$ by $\mathbf{Q}(\mathbf{x}_i) \in \mathbb{R}^{P,m}$ for each $i \in [n]$. Here, the p-th row of $\mathbf{Q}(\mathbf{x}_i)$ is $\mathbf{q}_p(\mathbf{x}_i)$. Using the same steps as in Section 5.1.1, the problem can be solved exactly as in Lemma 2, with a parameter matrix $\mathbf{A} \in \mathbb{R}^{m,Pd_2}$. There, each $\mathbf{A}_k$ is defined as $\mathbf{A}_k := \sum_{j=1}^{r} \mathbf{Q}^{\mathrm{T}} \mathbf{c}_j \boldsymbol{\alpha}_{k,j}^{\mathrm{T}}$ where $\mathbf{c}_j \in \mathbb{R}^{nP}$ is a vector whose (i,p)-th element is $c_{(i,p)}$ for the j-th filter.

5.3 Function classes

Our model class now corresponds to a collection of functions based on imposing constraints on the underlying matrix $\mathbf{A}$. In the case of multiple output nodes, we can define the function class $\mathcal{N}_c$, that is, the collection of functions based on the fully-connected shallow network class described in equation (4), but with the restriction that linear activation functions are used, such that a rank constraint is induced. This function class is thus defined as

$$\mathcal{N}_c(B_1, B_2) := \left\{ f^{\mathbf{A}} \text{ of the form (13)} : \|\mathbf{A}\|_* \leq d_2 B_1 B_2 r \right\} \tag{31}$$

for the convex relaxation of the shallow fully-connected neural network, and

$$\mathcal{F}^{l}_{\text{ccnn}}(B_1, B_2) := \left\{ f^{\mathbf{A}} \text{ of the form (15)} : \|\mathbf{A}\|_* \leq d_2 B_1 B_2 r \right\} \tag{32}$$

for the convex relaxation of the shallow convolutional neural network, when using the linear activation function. When using nonlinear activation functions as described in Section 5.2.2, there is a small change in the norm bound. As described in [1, section 3.2 and Appendix B], this is due to use of the kernel trick to handle the nonlinear activation functions. The CCNN function class using nonlinear activation functions is defined as:

$$\mathcal{F}_{\text{ccnn}}(B_1, B_2) := \left\{ f^{\mathbf{A}} \text{ of the form (15)} : \|\mathbf{A}\|_* \leq d_2 C_\sigma(B_1) B_2 r \right\}, \tag{33}$$

where C_σ is a monotonically increasing function which depends on the chosen kernel (See [1, Lemma 1 and Lemma 2]). Furthermore, we are guaranteed that $\mathcal{N}_c \supseteq \mathcal{N}$ and $\mathcal{F}_{\text{ccnn}} \supseteq \mathcal{F}_{\text{cnn}}$ [1].

5.4 Choice of kernel and activation functions

There are a few remarks we must mention before continuing. Firstly, steps in Section 5.2.2 hold only for certain activation functions and kernels. This section briefly summarizes the points made by [1] on this matter. Secondly, the above means that in practice we must calculate the matrix $\mathbf{Q}$, a question which we address in Section 6.

In Section 5.2 we stated that for the filters to be contained in the RKHS, that is, for Lemma 6 to hold, the right kernel function k and a sufficiently smooth activation function σ must be chosen. In [1] this point is expanded upon and here we will summarize their reasoning without repeating their proofs.

The first ingredient for Lemma 6 is a positive semidefinite kernel function $k : \mathbb{R}^{d_1} \times \mathbb{R}^{d_1} \to \mathbb{R}$ whose associated RKHS is "large enough" to contain any function of the form $h : \mathbf{z} \mapsto \sigma(\langle \mathbf{w}, \mathbf{z} \rangle)$ for particular activation functions σ (the "richness" requirement). [1] show how this richness requirement can be satisfied for the Inverse Polynomial kernel and the Gaussian RBF kernel.

Definition 3. The inverse polynomial kernel is defined as:

$$k^{\text{IP}}(\mathbf{z}, \mathbf{z}') := \frac{1}{2 - \langle \mathbf{z}, \mathbf{z}' \rangle}, \quad \|\mathbf{z}\|_2 \leq 1, \|\mathbf{z}'\|_2 \leq 1. \tag{34}$$

Definition 4. The Gaussian RBF kernel is defined as:

$$k^{\text{RBF}}(\mathbf{z}, \mathbf{z}') := \exp(-\gamma \|\mathbf{z} - \mathbf{z}'\|_2^2 \quad \|\mathbf{z}\|_2 = \|\mathbf{z}'\|_2 = 1, \gamma > 0. \tag{35}$$

They do this by (a) first verifying that these functions are indeed kernel functions and (b) proving that the associated RKHS contains the class of nonlinear filters. Verifying that the IP kernel and the Gaussian RBF functions are kernels is done by showing that there exists a mapping $\phi : \mathbb{R}^{d_1} \mapsto \ell^2(\mathbb{N})$ such that $k(\mathbf{z}, \mathbf{z}') = \langle \phi(\mathbf{z}), \phi(\mathbf{z}') \rangle$. They subsequently use polynomial expansions of $\sigma : \sigma(t) = \sum_{j=0}^{\infty} a_j t^j$ to show the second point.

The second ingredient for Lemma 2 is a sufficiently smooth activation function σ. In their work, [1] again define this criterion by using the polynomial expansion of σ. The rate at which the coefficients $\{a_j\}_{j=0}^{\infty}$ of the polynomial expansion of σ converges to zero is used as a measure of smoothness. Specifically, they mention:

1. arbitrary polynomial functions,
2. sinusoid activation functions $\sigma(t) = \sin(t)$,
3. erf function $\sigma_{\text{erf}} := 2/\sqrt{\pi} \int_0^t e^{z^2} dz$,
4. a smoothed hinge loss $\sigma_{\text{sh}} := \int_{\infty}^t \frac{1}{2}(\sigma_{\text{erf}}(z) + 1)dz$.

They state that different activation functions pair differently with different kernel functions. The main point they make is that the IP kernel captures all of the four mentioned above in its associated RKHS, the Gaussian kernel only the first two. Other activation functions which are popular in the field of deep learning, specifically, the ReLU activation function, are not smooth enough for either of those two kernels and as such cannot be used in the CCNN algorithm.

6 Learning shallow CCNNs

At this moment, we can describe an overview of the two-layer CCNN algorithm. The algorithm consists of four main steps which describe the order of operations between the input X and output Y. The goal is to learn the unknown function $f : X \mapsto Y$. The four steps are conceptually summarized in Algorithm 1. The four steps in the algorithm imply two practical complications. The first is the need to find a suitable approximation for the kernel matrix $\mathbf{K} \approx \mathbf{Q}\mathbf{Q}^{\mathrm{T}}$. The second is the question of how to optimize the network under a nuclear norm constraint. The following sections describe these issues in more detail.

6.1 Approximating the kernel matrix using the Nystrom method

An approximation $\mathbf{K} \approx \mathbf{Q}\mathbf{Q}^{\mathrm{T}}$ must be constructed for some choice of factorization method and some choice of m. Then $\mathbf{Q} \in \mathbb{R}^{nP,m}$. One such way is the *Nystrom method* [11]. Using the Nystrom method, an approximation $\hat{\mathbf{K}} = \mathbf{Q}\mathbf{Q}^{\mathrm{T}}$ is obtained by randomly sampling m rows/columns from the original $\mathbf{K}$. In our case, this means that the patch matrix of a data sample $\mathbf{Z}(\mathbf{x}_i) \in \mathbb{R}^{P,d_1}$ is represented by a approximate matrix $\mathbf{Q}(\mathbf{x}_i) \in \mathbb{R}^{P,m}$, with $\mathbf{Q}$ resulting from the approximation $\hat{\mathbf{K}}$. By [11] it is expected that good results can be obtained for $m \ll nP$, but whether this holds in our case is yet to be seen.

6.2 Projected gradient descent on the nuclear norm ball

In the third step of Algorithm 1, we face the constrained optimization problem

$$\hat{\mathbf{A}} = \underset{\|\mathbf{A}\|_* \leq R}{\arg\min} \quad \sum_{i=1}^{n} \mathcal{L}\Big(\mathrm{tr}(\mathbf{Q}(\mathbf{x}_i)\mathbf{A}_1), \ldots, \mathrm{tr}(\mathbf{Q}(\mathbf{x}_i)\mathbf{A}_{d_2}); y_i\Big). \tag{36}$$

input : Data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$, kernel function k, regularization parameter $R > 0$

1. Construct a kernel matrix $\mathbf{K} \in \mathbb{R}^{nP,nP}$ such that the entry at column (i,p) and row (i',p') is equal to $k(\mathbf{z}_p(\mathbf{x}_i), \mathbf{z}_{p'}(\mathbf{x}_{i'}))$. Compute a factorization $\mathbf{K} = \mathbf{Q}\mathbf{Q}^T$ or an approximation $\mathbf{K} \approx \mathbf{Q}\mathbf{Q}^T$, where $\mathbf{Q} \in \mathbb{R}^{nP,m}$.

2. For each $\mathbf{x}_i$, replace the patch matrix $\mathbf{Z}(\mathbf{x}_i) \in \mathbb{R}^{P,d_1}$ by $\mathbf{Q}(\mathbf{x}_i) \in \mathbb{R}^{P,m}$ whose p-th row is the (i,p)-th row of $\mathbf{Q}$.

3. Solve the following optimization problem to obtain a matrix $\hat{\mathbf{A}} := (\hat{\mathbf{A}}_1, \ldots, \hat{\mathbf{A}}_{d_2})$:

$$\hat{\mathbf{A}} \in \underset{\|\mathbf{A}\|_* \leq R}{\arg\min} \quad \tilde{\mathcal{L}}(\mathbf{A})$$

where

$$\tilde{\mathcal{L}}(\mathbf{A}) := \sum_{i=1}^n \mathcal{L}\Big(\big(\mathrm{tr}(\mathbf{Q}(\mathbf{x}_i)\mathbf{A}_1), \ldots, \mathrm{tr}(\mathbf{Q}(\mathbf{x}_i)\mathbf{A}_{d_2})\big); \mathbf{y}_i\Big).$$

output: Return the predictor $\hat{f}_{\mathrm{ccnn}}(\mathbf{x}) := \big(\mathrm{tr}(\mathbf{Q}(\mathbf{x})\hat{\mathbf{A}}_1), \ldots, \mathrm{tr}(\mathbf{Q}(\mathbf{x})\hat{\mathbf{A}}_{d2})\big)$.

Algorithm 1: *(Source: [1]) Learning two-layer CCNNs*

Here and in the sections that follow, we choose to use R as a shorthand for the nuclear norm bound, which in our case is $C_\sigma(B_1)B_2rd_2$. Generally, optimization problems can be solved by gradient descent, an iterative method where at each time point t, the coefficients at the next step are computed via $\mathbf{A}^{t+1} = \mathbf{A}^t - \eta\nabla\mathcal{L}(\mathbf{A}^t)$, where η is the step size and $\nabla\mathcal{L}(\mathbf{A}^t)$ the gradient of the loss function with respect to the parameters at time t. This step is then repeated until convergence. For our constrained problem in equation (36) however, the problem is that this update might result in next-step parameters $\mathbf{A}^{t+1}$ which do not adhere to the constraint $\|\mathbf{A}\|_* \leq R$, because the optimal solution at that iteration may lie outside the constraint set.
One way to solve this issue is through the use of projected gradient descent, in which the update step is replaced by

$$\mathbf{A}^{t+1} = \prod_R \Big(\mathbf{A}^t - \eta\nabla\mathcal{L}(\mathbf{A}^t)\Big), \tag{37}$$

where $\prod_R(\mathbf{A})$ denotes the Euclidean projection of $\mathbf{A}$ on the nuclear norm ball with radius R. The necessary steps are described from a top-down perspective below, and is a combination of work by [12] and [13].

Lemma 7 (Projecting a matrix on the nuclear norm)**.** *(Source: [12, Section 3.3] and [13]). Let* $\mathbf{A}$ *be a matrix with a bounded nuclear norm* $\|\mathbf{A}\|_* \leq R$ *and let* $\boldsymbol{\sigma}$ *be the vector of the singular values of* $\mathbf{A}$*, such that* $(\boldsymbol{\sigma})_i = \sigma_i$*. Projecting* $\mathbf{A}$ *on the nuclear norm ball with radius* R*,* $\{\hat{\mathbf{A}} : \|\hat{\mathbf{A}}\|_* \leq R\}$ *is equivalent to solving the constrained optimization problem*

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^n} \|\boldsymbol{\beta} - \mathbf{u}\|_2^2 \quad s.t. \quad \sum_{i=1}^n \beta_i = R \quad and \quad \beta_i \geq 0, \tag{38}$$

then setting $\sigma_i^* = sign(\sigma_i)\beta_i$*, and then calculating*

$$\hat{\mathbf{A}} = \mathbf{U}\Sigma_A^*\mathbf{V}^T \quad where \quad \Sigma_A^* = diag(\boldsymbol{\sigma}_A^*), \tag{39}$$

with $\boldsymbol{\sigma}$*,* $\mathbf{U}$ *and* $\mathbf{V}$ *being the singular values, the left-singular vectors and right-singular vectors of* $\mathbf{A}$*, respectively, and where for each* i*,* $u_i = |\sigma_i|$*.*

The proof to Lemma (7) requires several steps which are, in order:

1. Projection of a vector on the simplex (see Section 6.2.1);
2. Projection of a vector on the ℓ_1-ball which can be re-written into the simplex problem from step 1 (see Section 6.2.2);
3. Projection of a matrix on the nuclear norm ball, which can be solved by projecting the singular values of said matrix on the ℓ_1-ball, and then reconstructing the matrix using the projected singular values.

6.2.1 Projecting the singular values on the simplex

First, consider the problem of projecting the singular values on the simplex. As adapted from [13], this can be written as the following constrained optimization problem:

$$\min_{\boldsymbol{\sigma}^*} \frac{1}{2}\|\boldsymbol{\sigma}^* - \boldsymbol{\sigma}\|_2^2 \quad \text{s.t.} \quad \sum_{i=1}^{n} \sigma_i^* = R, \quad \sigma_i^* \geq 0, \tag{40}$$

where $\boldsymbol{\sigma}$ denotes the vector of singular values that must be projected on the simplex, $\boldsymbol{\sigma}^*$ the vector of singular values on the simplex set.

Lemma 8 (Projecting singular values on the simplex)**.** *(Based on [13]). Suppose we are given a vector of singular values $\boldsymbol{\sigma}$ that must be projected on the simplex radius R. Then, solving optimization problems of the form (40) is equivalent to calculating the elements of $\boldsymbol{\sigma}^*$ as*

$$\sigma_i^* = \max\{\sigma_i - \theta, 0\}, \tag{41}$$

where

$$\theta = \frac{1}{\rho}\Big(\sum_{i=1}^{\rho} \mu_i - R\Big), \tag{42}$$

and

$$\rho(R, \boldsymbol{\mu}) = \max\left\{ j \in [n] : \mu_j - \frac{1}{j}\left(\sum_{m=1}^{j} \mu_r - R\right) > 0 \right\}, \tag{43}$$

where $\boldsymbol{\mu}$ denotes the vector obtained by sorting $\boldsymbol{\sigma}$ in a descending order. In equation (42) ρ is used as a shorthand for $\rho(R, \boldsymbol{\mu})$ from equation (43).

Proof. The proof for Lemma 8 uses Lemmas 9 and 10 below. First, we begin the proof by re-writing (40) into the Lagrangian dual form:

$$L(\boldsymbol{\sigma}^*, \boldsymbol{\lambda}, \boldsymbol{\theta}) = \frac{1}{2}\sum_{i=1}^{n}\Big(\sigma_i^* - \sigma_i\Big)^2 - \sum_{i=1}^{n}\lambda_i\sigma_i^* + \theta\Big(\sum_{i=1}^{n}\sigma_i^* - R\Big) \tag{44}$$

on the account that we can recognize the n inequality constraints $g_i(\boldsymbol{\sigma}^*) = -\sigma_i^*$ and the single equality constraint $h(\boldsymbol{\sigma}^*) = \sum_{i=1}^{n}\sigma_i^* - R$. Finding the optimal values $\boldsymbol{\sigma}^*$ involves taking the derivative of (44) w.r.t. σ_i^* and setting to zero. Thus

$$\frac{\partial L(\boldsymbol{\sigma}^*, \boldsymbol{\lambda}, \boldsymbol{\theta})}{\partial \sigma_i^*} = \sigma_i^* - \sigma_i - \lambda_i + \theta = 0 \leftrightarrow \sigma_i^* = \sigma_i + \lambda_i - \theta. \tag{45}$$

Since $\sigma_i^* \geq 0$ (because of the constraint in problem (40)) and the fourth KKT condition (see Definition 5 in the Appendix), we know that if $\sigma_i^* > 0$, then $\lambda_i = 0$. Therefore it follows that

$$\sigma_i^* = \max\{\sigma_i - \theta, 0\}. \tag{46}$$

We now need to find the value of θ. If there would only be nonzero elements in $\boldsymbol{\sigma}^*$, we could simply find it via

$$\theta = \frac{1}{n}\Big(\sum_{i=1}^{n}\sigma_i - R\Big), \tag{47}$$

but the current issue is that there are nonzero elements in $\boldsymbol{\sigma}^*$ and we do not know the indices of the corresponding elements in $\boldsymbol{\sigma}$. In order to know which ones they are, we can use the following lemma:

Lemma 9. *(Source: [12, Lemma 2], [13, Lemma 1]). Let $\boldsymbol{\sigma}^*$ be the optimal solution to a minimization problem of the form (40). Let s and j be two indices such that $\sigma_s > \sigma_j$. If $\sigma_s^* = 0$ then σ_j^* must be zero as well.*

From this lemma we know that $\sum_{i=1}^{\rho} \sigma_{(i)}^* = \sum_{i=1}^{\rho} \sigma_{(i)} - \theta$ and $\sum_{i=\rho+1}^{n} \sigma_{(i)}^* = \sum_{i=\rho+1}^{n} \sigma_{(i)} + \lambda_{(i)} - \theta = 0$, where $\sigma_{(i)}$ refers to the i-th element of the vector $\boldsymbol{\sigma}$ if its elements are sorted in descending order. We just need to know ρ, the number of nonzero elements in $\boldsymbol{\sigma}^*$. For this, we use the following lemma:

Lemma 10 (Finding ρ)**.** *(Source: [12, Lemma 3], [13, Lemma 2]). Let $\boldsymbol{\sigma}^*$ be the optimal solution to the minimization problem given in equation (40). Let $\boldsymbol{\mu}$ denote the vector obtained by sorting $\boldsymbol{\sigma}$ in a descending order. Then, the number of strictly positive elements in $\boldsymbol{\sigma}^*$ is*

$$\rho(R, \boldsymbol{\mu}) = \max \left\{ j \in [n] : \mu_j - \frac{1}{j} \left(\sum_{r=1}^{j} \mu_r - R \right) > 0 \right\}. \tag{48}$$

We can now find ρ using equation (43) and then θ as follows:

$$\theta = \frac{1}{\rho} \Big(\sum_{i=1}^{\rho} \sigma_{(i)} - R \Big) \tag{49}$$

where ρ is the number of nonzero elements in σ^*, and $\sigma_{(i)}$ denotes the i-th element in the vector $\boldsymbol{\sigma}$ if it is sorted in descending order. Then, we can calculate the projection of $\boldsymbol{\sigma}$ on the simplex via equation (46). □

6.2.2 Projecting the singular values on the ℓ_1-ball

Now that we can project singular values on the simplex, we can show that projecting them on the ℓ_1-ball is but a small step away. We show how to turn that problem into the simplex projection. If we define the projection on the ℓ_1-ball as the following constrained optimization problem:

$$\min_{\boldsymbol{\sigma}^* \in \mathbb{R}^n} \|\boldsymbol{\sigma}^* - \boldsymbol{\sigma}\|_2^2 \quad \text{s.t.} \quad \|\boldsymbol{\sigma}^*\|_1 \leq R. \tag{50}$$

we can use the following lemma to re-write the problem into the simplex case (40).

Lemma 11. *(Source: [13]). If $\boldsymbol{\sigma}^*$ is an optimal solution to the problem in (50), then, for all i, $\sigma_i^* \sigma_i \geq 0$.*

Combining the above, we can now state the method to project a vector on the ℓ_1-ball. When presented with a minimization problem of the form (40), define $\mathbf{u}$ such that for each i, $u_i = |\sigma_i|$. Then, solve the following problem by projecting $\mathbf{u}$ on the simplex, as described in the previous section. The problem will take this form:

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \|\boldsymbol{\beta} - \mathbf{u}\|_2^2 \quad \text{s.t.} \quad \sum_{i=1}^{n} \beta_i = R \quad \text{and} \quad \beta_i \geq 0, \tag{51}$$

and once the solution to this problem is obtained, the problem of projecting on the ℓ_1-ball (40) can be solved by setting $\sigma_i^* = \text{sign}(\sigma_i)\beta_i$. Now that we know how to project a vector on the ℓ_1-ball, we can project a matrix on the nuclear norm ball by projecting its singular value on the ℓ_1-ball; given a matrix $\mathbf{A}$ that needs projecting on the nuclear norm ball, we can project its singular values $\boldsymbol{\sigma}_A$ on the ℓ_1-ball as just described, and then reconstruct the projected matrix as in equation (39). This means that we now know how to solve optimization problems of the form (36).

6.3 Algorithm

Putting the steps described in the previous section together we can define an algorithm to project a matrix $\mathbf{A}$ on the nuclear norm ball such that $\|\mathbf{A}\|_* \leq R$, and then incorporate that into the gradient descent algorithm. At each timestep t, the new parameters are projected on the nuclear norm ball. The steps are shown in Algorithm 2 below.

input : A matrix $\mathbf{A}$ and a scalar $R > 0$
Compute the SVD: $\mathbf{A} = \mathbf{U}\Sigma_A\mathbf{V}^{\mathrm{T}}$ and let $\boldsymbol{\sigma}$ be the vector of singular values;
Define $\mathbf{u}$ s.t. $u_i = |\sigma_i|$;
Sort $\mathbf{u}$ into $\boldsymbol{\mu} : \mu_1 \geq \mu_2 \geq \cdots \geq \mu_p$;
Find $\rho(R, \boldsymbol{\mu}) = \max\left\{ j \in [n] : \mu_j - \frac{1}{j}\left(\sum_{r=1}^{j} \mu_r - R \right) > 0 \right\}$;
Define $\theta = \frac{1}{\rho}\left(\sum_{i=1}^{\rho} \mu_i - R \right)$;
Calculate $\boldsymbol{\beta}$ s.t. $\beta_i = \max\{u_i - \theta, 0\}$;
Calculate $\boldsymbol{\sigma}^*$ s.t. $\sigma_i^* = \text{sign}(\sigma_i)\beta_i$ and construct from these the diagonal matrix $\boldsymbol{\Sigma}_A^*$;
output: $\hat{\mathbf{A}} = \mathbf{U}\Sigma_A^*\mathbf{V}^{\mathrm{T}}$

Algorithm 2: *Algorithm for projection of a matrix on the nuclear norm ball.*

input : Data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$, kernel function k, regularization parameter $R > 0$, number of filters r, stropping criterion ϵ
while $L(\mathbf{A}_{t-1}) - L(\mathbf{A}_t) > \epsilon$ **do**
  not yet converged;
  Choose a stepsize; we will be using a constant stepsize η;
  Calculate gradients using stochastic gradient descent (see also Section 4.2);
  Calculate indermediary update $\mathbf{A}'_{t+1} = \mathbf{A}_t - \eta_t \nabla \mathcal{L}(\mathbf{A}_t)$;
  Update coefficient matrix $\mathbf{A}_{t+1} = \prod_R(\mathbf{A}'_{t+1})$ using Algorithm 2;
end

Algorithm 3: *Projected gradient descent of a matrix on the nuclear norm ball*

7 Applications

In order to test the performance of the CCNN algorithms on relatively easy datasets, data was generated according to a *sum of sigmoids* model and a *radial function* model. Inspiration for this data generation mechanism was drawn from [14]. For a more complicated data set, data in the form of text was chosen.

7.1 Simulated data

First, random predictor data was generated from a multivariate standard normal distribution: $\mathbf{X} \in \mathbb{R}^{n,p} \sim N(\mathbf{0}, \mathbf{I}_p)$, where n indicates the number of samples and p the dimension of the predictor data. Then, for the sum of sigmoids model, observations $\mathbf{y}_s$ were generated via the nonlinear function

$$\mathbf{y}_s = f_s(\mathbf{X}) = \sum_{i=1}^{m} \sigma(\mathbf{X}\boldsymbol{\beta}_i), \quad \text{where } \sigma(\mathbf{t}) = \frac{1}{1+\exp(-\mathbf{t})}. \tag{52}$$

Here we chose to set $m = 2$, $\boldsymbol{\beta}_1 = (3,3)^T$, and $\boldsymbol{\beta}_2 = (3,-3)^T$. For the radial model, data was generated via the nonlinear function

$$\mathbf{y}_r = f_r(\mathbf{X}) = \prod_{j=1}^{p} \psi(\mathbf{X}_{(\cdot,j)}), \quad \text{where } \psi(\mathbf{t}) = \left(\frac{1}{2\pi}\right)^{1/2} \exp(-(\mathbf{t}^{\mathrm{T}}\mathbf{t})/2). \tag{53}$$

Since the CCNN algorithm puts heavy emphasis on classification, for both sets of simulated data, a noisy binary version was constructed. First, for the sum of sigmoid model and for the radial basis function model, errors $\boldsymbol{\epsilon}_s$ and $\boldsymbol{\epsilon}_r$ were sampled from a normal distribution with a variance $\mathrm{Var}(\boldsymbol{\epsilon})$ scaled such that the ratio of the true and error variance was 4:

$$\frac{\mathrm{Var}(\mathrm{E}(\mathbf{y}|\mathbf{X}))}{\mathrm{Var}(\mathbf{y} - \mathrm{E}(\mathbf{y}|\mathbf{X}))} = \frac{\mathrm{Var}(f(\mathbf{X}))}{\mathrm{Var}(\boldsymbol{\epsilon})} = 4. \tag{54}$$

Then, the observed (noisy) binary outcomes $\tilde{\mathbf{y}}_s$ and $\tilde{\mathbf{y}}_r$ were generated as follows:

$$\tilde{\mathbf{y}}_s = \begin{cases} 1 & \text{if } \mathbf{y}_s + \boldsymbol{\epsilon}_s \geq \bar{\mathbf{y}}_s \\ 0 & \text{if } \mathbf{y}_s + \boldsymbol{\epsilon}_s < \bar{\mathbf{y}}_s \end{cases} \tag{55}$$

and

$$\tilde{\mathbf{y}}_r = \begin{cases} 1 & \text{if } \mathbf{y}_r + \boldsymbol{\epsilon}_r \geq \bar{\mathbf{y}}_r \\ 0 & \text{if } \mathbf{y}_r + \boldsymbol{\epsilon}_r < \bar{\mathbf{y}}_r \end{cases} \tag{56}$$

that is to say, the observation $\tilde{\boldsymbol{y}} = 1$ if the real signal plus noise is higher than the average of all real signals, and 0 otherwise. Figure 4 shows the simulated data sets for the sum of sigmoids and radial models.

7.1.1 Implementation details

For both sets of generated data, baseline models of SVM, KNN, Logistic regression, and fully-connected neural networks were modeled in order to compare the results with the best possible CCNN. Since in practice deep learning specialists

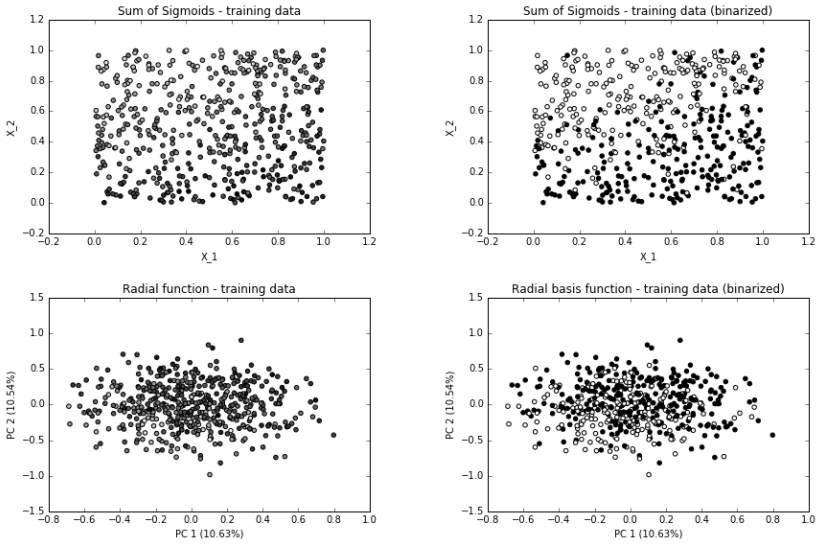


Figure 4: *Visualizations of the simulated datasets in 2 dimensions. For the radial function, PCA was applied such that 2-dimensional visualization is possible.*

fine-tune their models to achieve better results, a simple tuning mechanism was applied by running models for different numbers of nodes in the hidden layer of the network: $5, 25$ and 100, respectively. For the CCNN implementation, experiments were done with different Nystrom dimensions $m \in \{1, 2, 5, 25\}$) (see Section 6.1) and regularization parameter values for the nuclear norm constraint $R \in \{0.01, 0.1, 1, 10\}$). This was done because it is believed that these parameters will influence the training and accuracy of the final CCNN model.

7.1.2 Results

The results of the fully-connected shallow neural network and CCNN models are shown in Figures 6, 7, 8 and 9. The accuracies of all models are summarized in Table 2. For both the sum of sigmoids and radial function generated datasets, the best CCNN achieved accuracies comparable to the best alternatives, with classification accuracies of 83.1% and 78.28% for the sum of sigmoids and radial function sets, respectively.

7.2 Text classification

Now we describe the application of the CCNN model on a real data set consisting of a text classification problem. Clickbait is a term for web-based content which aims to convince users to click on said content and browse to the advertised website. Clickbait is often used by websites which aim to generate revenue by spreading these types of texts on social media. The clickbait headlines typically aim to exploit the so-called "curiosity gap" by including certain terms that are psychologically stimulating the reader into desiring to know more. The type of language used in clickbait headlines is markedly different from serious headlines, but it can be difficult to concisely describe these differences of to

make a list of words which are typical for clickbait content. This provides us with an interesting proposition: whether it is possible for a neural network to distinguish between clickbait versus non-clickbait content by learning some underlying function of the words used in the headlines.

In order to do this, it is necessary to have a dataset of clickbait and non-clickbait content such that various models can be trained to classify among the two categories. A dataset of 32,000 headlines was made available by [15], who have used it in their paper [16] which shows how several models (SVM, Decision Trees, Random Forest) can be used to reasonably good effect for the classification task.

The clickbait corpus consists of article headlines from *BuzzFeed*, *Upworthy*, *ViralNova*, *Thatscoop*, *Scoopwhoop* and *ViralStories*. The non-clickbait article headlines are collected from *WikiNews*, *New York Times*, *The Guardian*, and *The Hindu*.

7.2.1 Representing text data using tokens

Suppose we have data $\{\mathbf{t}_i, y_i\}_{i=1}^n$ on n sequences of text. Here, each sequence $\mathbf{t}_i = (t_{i1}, t_{i2}, \ldots, t_{iJ})$ is assumed to be padded to the same length of J *tokens*. A *token* refers to a word, partial word or punctuation mark in the sequence. Because learning algorithms take numerical values as input, we must represent our sequences using numbers. The simplest way is to number all the tokens appearing in some given *vocabulary*. For example, the sequence `wait for the video and don't rent it` might be represented by $(3532, 7, 3, 150, 8, 383, 4676, 44)$. It is then padded on either side by a placeholder token such that it is of length J.

y_i represents some characteristic of the i-th sequence. For a continuous $y_i \in \mathbb{R}$, this could be a measure of sentiment of the sample $\mathbf{t}_i$. When the outcome is discrete, that is, $y_i \in \{1, 2, \ldots, G\}$ it could be a classification for $\mathbf{t}_i$, such as the type of text document among G categories.

7.2.2 Representing text data using word embeddings

When analyzing data of the form $\{\mathbf{t}_i, y_i\}_{i=1}^n$ described above, we aim to learn the function $f : T \mapsto Y$. Learning algorithms typically do this by using some sort of distance metric between the samples in a training data set. Given the tokenized version of text data, it is not immediately straightforward how the numerical representations for the tokens can be efficiently used for this. For this to work, tokens representing similar sentiments, indicators for document class, etc, should be assigned similar numerical values such that the distance between them is small. Vector methods give us the tools of distances or angles between pairs of words and pairs of sequences.

If each token in the vocabulary set would not be represented by a token $t_{ij} \in \mathbb{N}$ but by some higher-dimensional representation $\mathbf{t}_{ij} \in \mathbb{R}^u$, such that each sequence in a data set can be represented by a sequence of vectors:

$$\mathbf{T}_i = (\mathbf{t}_{i1}, \mathbf{t}_{i2}, \ldots, \mathbf{t}_{iJ}).$$

Then, the similarities and distances between words $k_1(\mathbf{t}_{ij}, \mathbf{t}_{i'j'})$ and sequences $k_2(\mathbf{T}_i, \mathbf{T}_{i'})$ can be computed for some suitably chosen distance function k_1 :

$\mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$ and $k_2 : \mathcal{V} \times \mathcal{V} \mapsto \mathbb{R}$. Here, $\mathcal{U}$ would be the vector space of the words, and $\mathcal{V}$ the matrix space of sequences.

In Figure 5 we can see that each of the tokens in the sequence `wait for the video and don't rent it` is represented by a vector, except `don't`, which is split into two vectors, `do` and `n't`. This is done such that any learning algorithm applied to such sequences may (hopefully) learn the distinct 'negative' characteristic of the token `n't`.

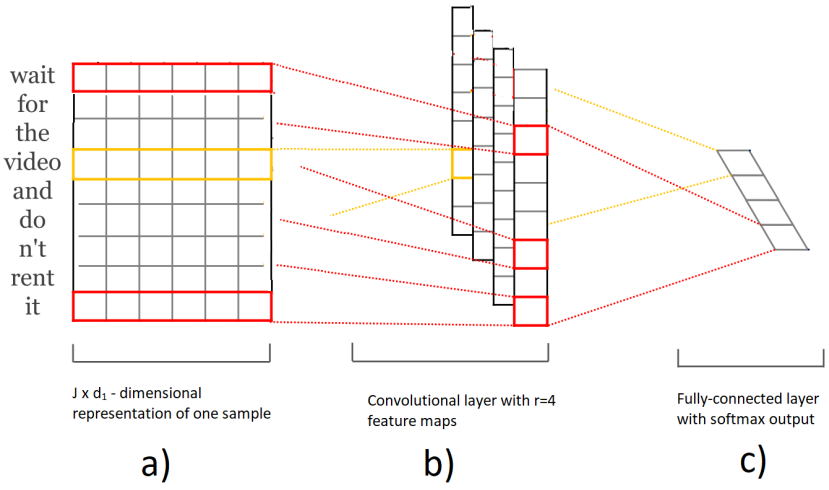


Figure 5: *(Adapted from [17]) Visualization of equations (6) and (7) applied to text document classification. Here, J refers to the sequence length (not to be confused with lowercase $j \in [r]$). d_1 is the word embedding dimension, r the number of filters in the CNN, and d_2 the number of output nodes. a): Input matrix $\mathbf{T} \in \mathbb{R}^{J,d_1}$. The red and yellow areas are treated as patch vectors $\mathbf{z}(\mathbf{x}) \in \mathbb{R}^{d_1}$. b): $h(\mathbf{z}(\mathbf{x}))$ filter activations $\in \mathbb{R}^J$ resulting from r filters with weights $\mathbf{w} \in \mathbb{R}^{d_1}$. c): output vector $\mathbf{y} \in \mathbb{R}^{d_2}$. Here, $d_2 = 4$ which happens to be equal to r in this particular example. The output layer uses coefficients α_{kjp} to learn the contribution of the p-th token through the j-th filter to the k-th output.*

One problem with this approach is how to find appropriate vector representations for the tokens. Ideally, each token in a vocabulary is assigned a representation such that its distance with tokens of similar meanings is small, and its distance with tokens if dissimilar meanings is high. Two methods to find appropriate token representations (from now on referred to as *word embeddings*) are Word2Vec [18] and Global Vectors (GloVe) [19]. The details of how they work are outside the scope of this thesis. Here, we choose to use GloVe, which provides pre-trained word embeddings for several different dimensions. In this thesis, we have chosen for the $u = 50$-dimensional embeddings.

7.2.3 Implementation details

It is these vectorized representations of the Clickbait data set which were used as input for the CNN and CCNN models. Each vectorized token in each headline is treated as a patch for the (convexified) convolutional neural network. Thus,

$\mathbf{T}_i = \mathbf{Z}(\mathbf{x}_i)$ and $\mathbf{t}_{ip} = \mathbf{z}_p(\mathbf{x}_i)$, when referring back to the notation used in Section 3.2.

As explained for the neural networks used on the simulated data, in practice the CNNs are finetuned with many hyperparameters and layers with extra functions in order to improve performance. In order to get a more fair comparison, the CNNs discussed here only use a *global max pooling layer* after the convolutional layer. The convolutional layer used 4, 16 and 64 filters, such that three different CNNs could be compared. In all models the filters used *ReLu* activation with a *softmax* activation in the output layer. The models were trained using *categorical cross-entropy* loss and *rmsprop* optimizer.

The CCNN model was trained in 16 different configurations for all combinations of four values for Nystrom dimension m and nuclear norm radius R. The Nystrom dimensions (see Section 6.1) used were $m \in \{5, 25, 100, 200\}$ and the radii were $R \in \{0.1, 1, 5, 100\}$. The loss function used was softmax and optimization was performed by *projected stochastic gradient descent*.

Alternative classification methods were also considered, with Logistic Regression, SVM and K-nearest neighbors being applied on the vectorized clickbait data.

All models were trained on 32,000 samples of which half were used for training and the other half reserved for validation.

7.2.4 Results

The results of the Clickbait data set analysis using the non-convex CNN models is shown in Figure 10. The benefit of using pre-trained word embeddings is clear from the high accuracy from the start, which only improves for the training sample and stays the same for the validation sample. Best results are obtained with 16 or 64 filters, where the validation accuracy reaches around 96%. While the training accuracy still has room for growth, from the loss plots in Figure 10 it shows that this is due to overtraining, since the validation loss is increasing after the 5th or so epoch, and the validation accuracy does not keep increasing.

The results of the CCNN models is shown in Figure 11. From these figures the effects of different values for m and R are clearly visible. Regarding nuclear norm radius, we can see in the loss plots that setting it to small values $0.1, 1$ or 5 makes it difficult to update the model in terms of loss, which stays quite high during the entire observed computational time. The effect of this is visible in the right-hand side plots in Figure 11, showing the validation accuracy. For these small values of R, the accuracy stays comparably bad (around the 65-70% range). Only for a radius of 5 it is possible to catch up to the best model when larger Nystrom dimensions m are used. Regarding the different values of Nystrom dimension m, we can also see the phenomenon that generally, higher values result in higher accuracy values on the validation set. Generally, using a high-dimensional Nystrom approximation and allowing for a large value of R result in a validation accuracy of close to 90%.

Finally, the SVM achieved a 92.57% accuracy, the KNN achieved 83.4% and logistic regression 91.36%. The results are summarized in Table 2.

Table 1: Main results

	Sum of sigmoids	Radial function	Clickbait
SVM	83.92%	79.49%	92.57%
KNN	81.79%	74.04%	83.40%
Logistic Regression	83.61%	78.49%	91.36%
NN-5	83.87%	78.09%	
NN-25	83.05%	78.25%	
NN-100	83.25%	78.17%	
CCN-4			92.27%
CCN-16			95.66%
CCN-64			95.44%
Best CCNN	83.1%	78.28%	90.02%

Table 2: *Classification accuracies of the models in this thesis. All data were analysed using SVM, KNN and Logistic regression. The simulated data contained no structure of hierarchical features and were thus assessed only using fully-connected neural networks to compare with the CCNN. The Clickbait data was analysed with CNNs and CCNNs.*

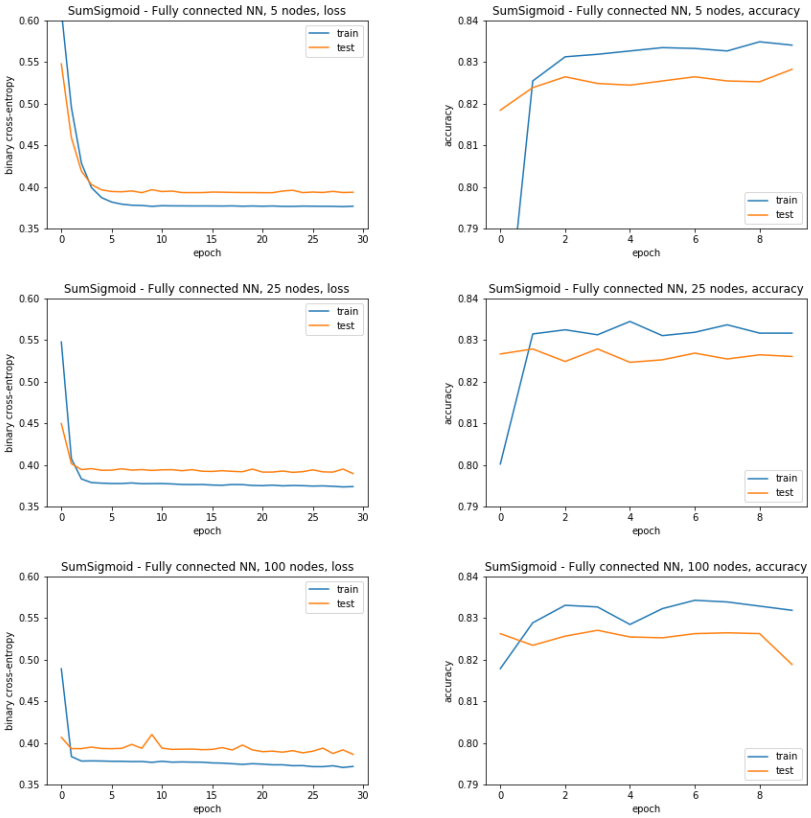


Figure 6: *Training results for the sum-of-sigmoids simulated data by three fully-connected neural networks (non-convexified) with 1 hidden layer and either 5, 25 or 100 hidden nodes. For categorization, loss was binary cross-entropy and accuracy the percentage correctly classified. Despite the fact that the data are not linearly separable in lower dimensions (see Figure 4), a very simple neural network achieves an 85% accuracy after 10 epochs.*

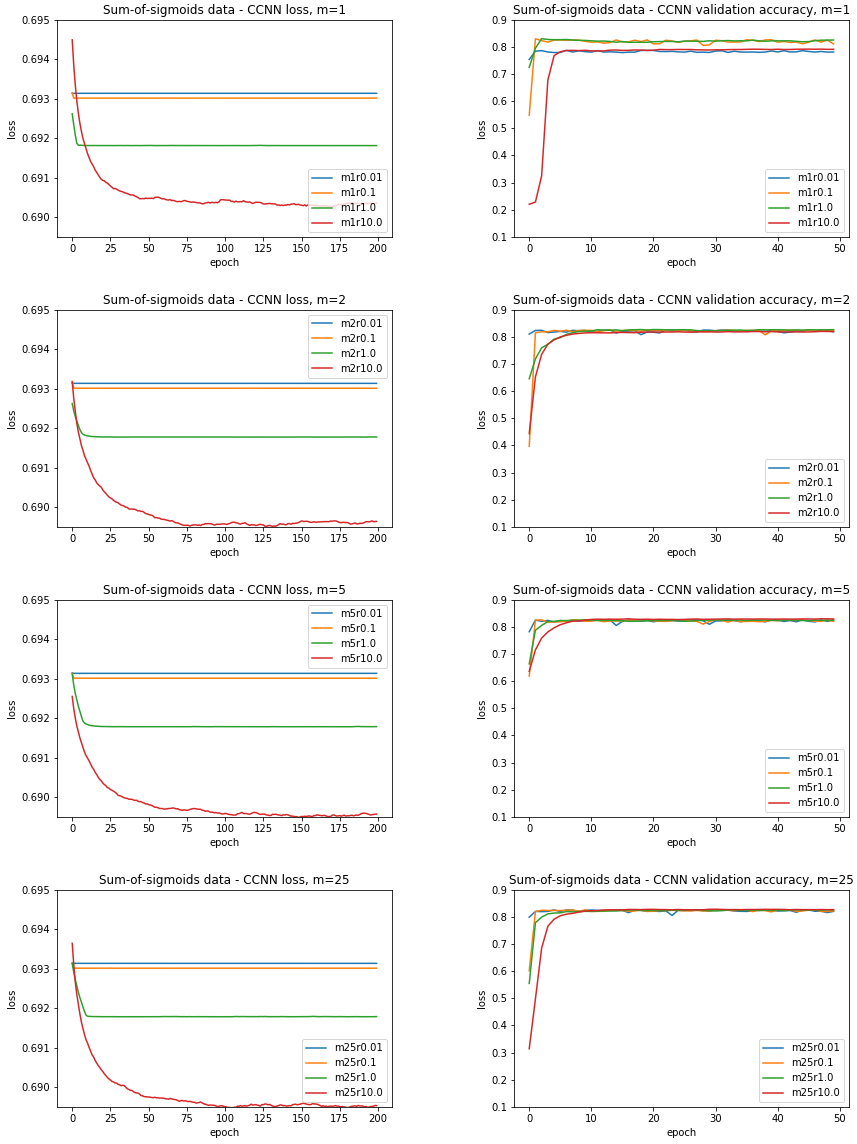


Figure 7: *Training results for the sum-of-sigmoids simulated data by shallow convexified convolutional neural networks. 16 models were trained for the different combinations for Nystrom dimension* $m \in \{1, 2, 5, 25\}$ *and nuclear norm bound* $\|\hat{\mathbf{A}}\|_* = R \in \{.01, .1, 1, 10\}$. *For categorization, loss was binary cross-entropy and accuracy the percentage correctly classified.*

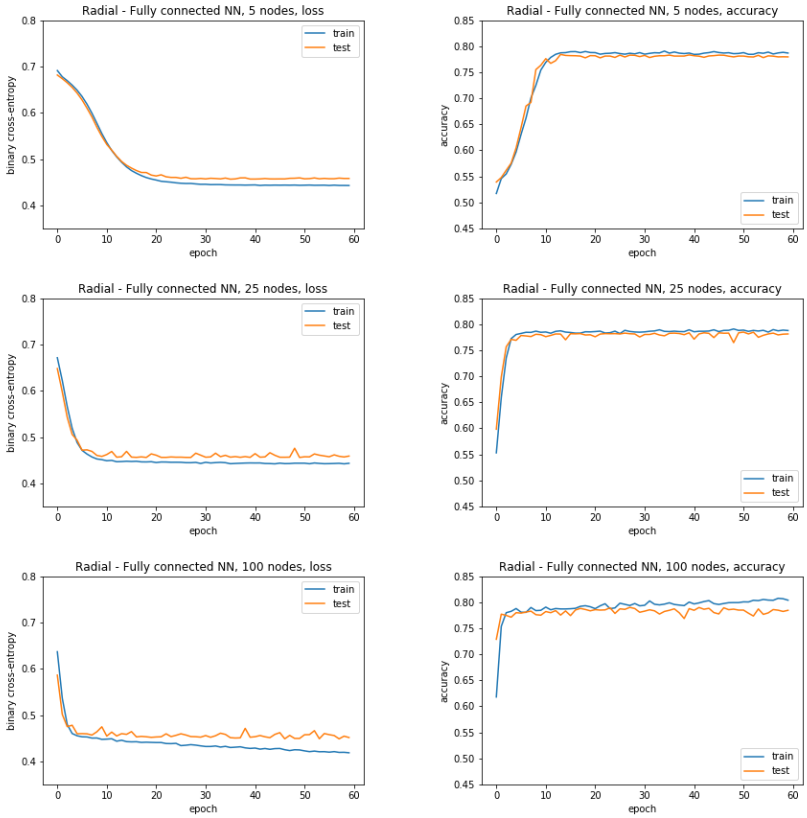


Figure 8: *Training results for the radial model simulated data by three fully-connected neural networks with 1 hidden layer and either 5, 25 or 100 hidden nodes. For categorization, loss was binary cross-entropy and accuracy the percentage correctly classified. Despite the fact that the data are not linearly separable in lower dimensions (see Figure 4), a very simple neural network achieves an 85% accuracy after 10 epochs.*

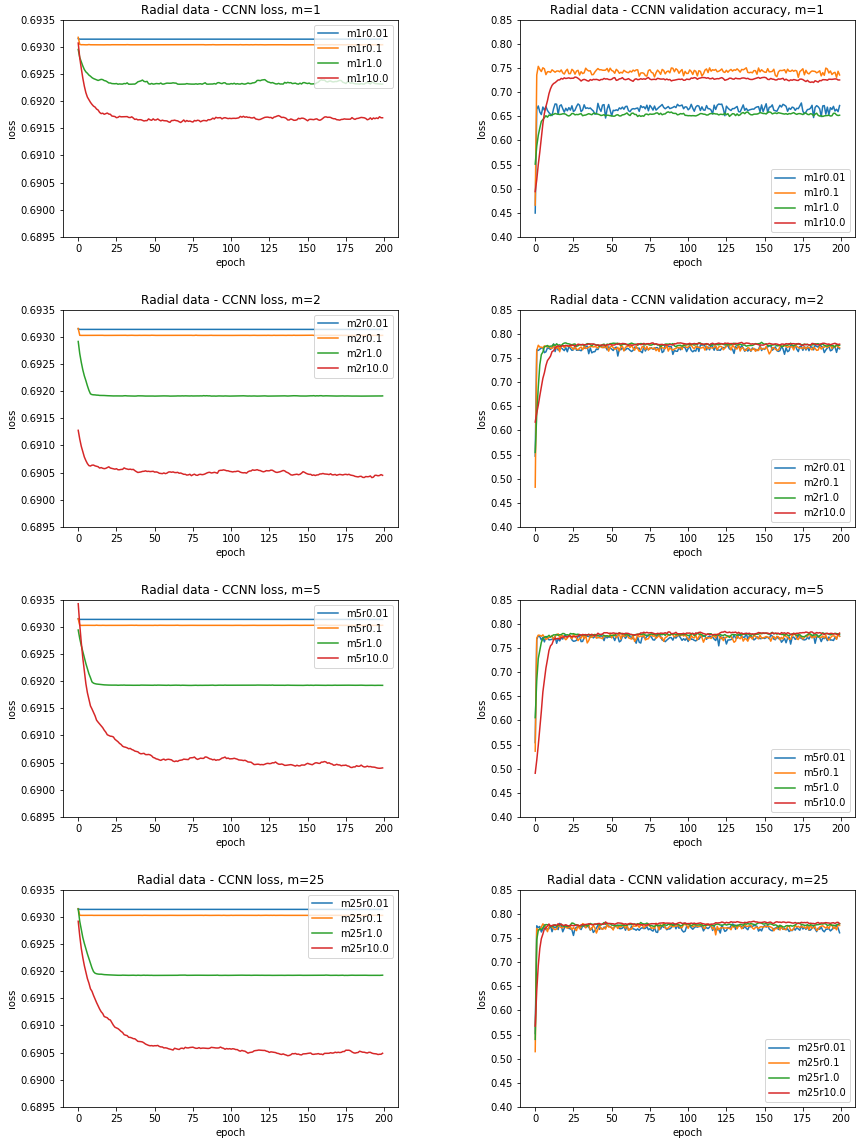


Figure 9: *Training results for the radial simulated data by shallow convexified convolutional neural networks. 16 models were trained for the different combinations for Nystrom dimension* $m \in \{1, 2, 5, 25\}$ *and nuclear norm bound* $\|\hat{\mathbf{A}}\|_* = R \in \{.01, .1, 1, 10\}$. *For categorization, loss was binary cross-entropy and accuracy the percentage correctly classified.*

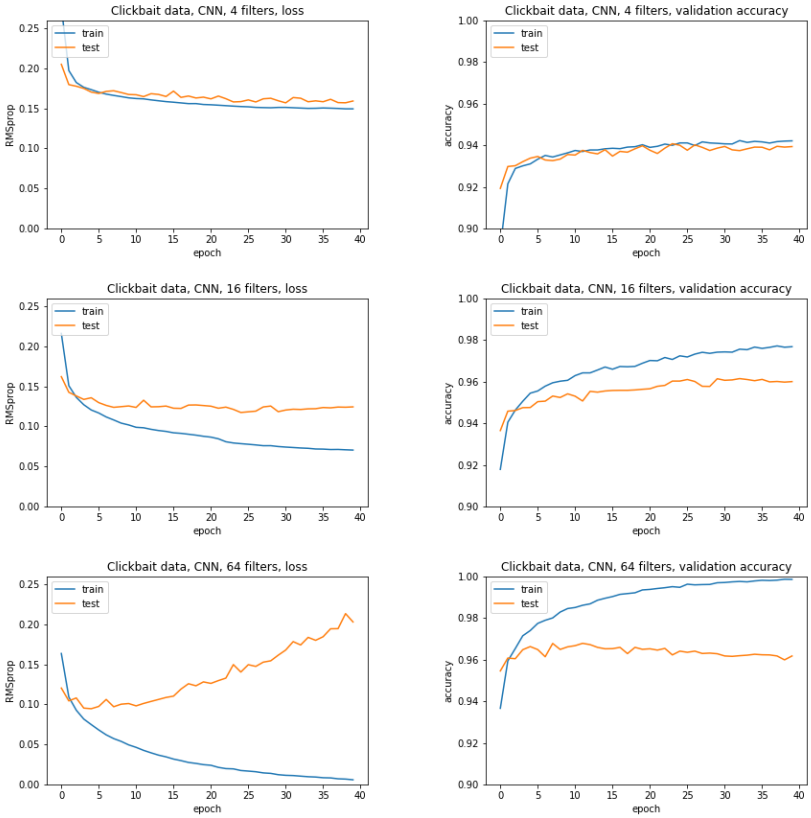


Figure 10: *Model performance for the clickbait task using a convolutional neural network with 1 hidden layer using $r \in \{4, 16, 64\}$ filters. The model was run for 40 iterations. The benefit of using pre-trained word embeddings is clear from the high accuracy from the start, which only improves for the training sample and stays the same for the validation sample. Best results are obtained with 16 or 64 filters, where the validation accuracy reaches around 96%. While the training accuracy still has room for growth, from the loss plots it shows that this is due to overtraining.*

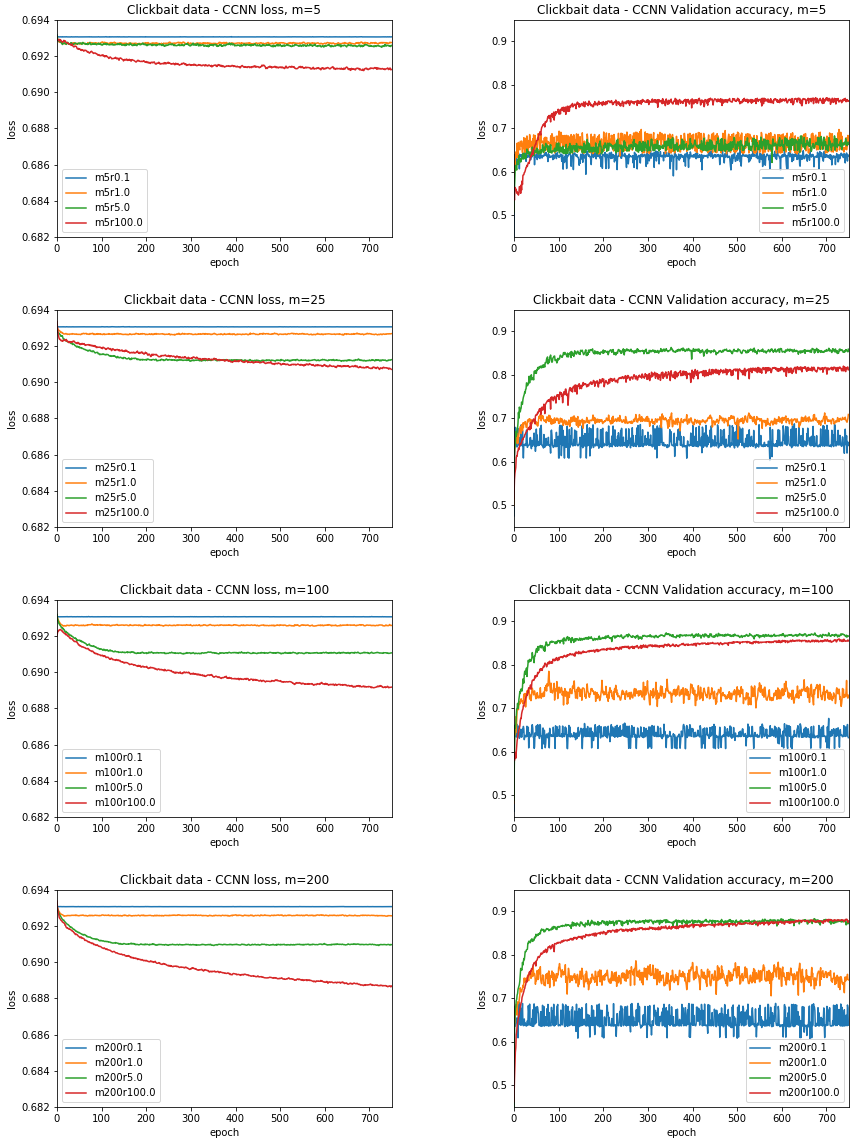


Figure 11: *Training results for the Clickbait data by shallow convexified convolutional neural networks. 16 models were trained for the different combinations for Nystrom dimension* $m \in \{5, 25, 100, 200\}$ *and nuclear norm bound* $\|\hat{\mathbf{A}}\|_* = R \in \{.01, .1, 1, 100\}$. *For categorization, loss was binary cross-entropy and accuracy the percentage correctly classified.*

8 Conclusion and discussion

In this paper, we have introduced the concepts of neural networks. We then introduced the concept of parameter sharing and how that leads to the convolutional neural network, a model architecture typically applied to classification tasks in image datasets. The learning algorithm implied by the neural network architecture makes use of back-propagation, in which the model coefficients are iteratively updated. This is a nonconvex optimization problem that is known to be NP hard. In practice this results in models whose performance is determined by the combination of a chosen model architecture, a set of hyperparameters such as number of hidden nodes, and a chosen optimization algorithm. In [1], a new model class known as *convexified convolutional neural networks* (CCNNs) was proposed. They show that training a CCNN corresponds to a convex optimization problem which can be solved by a projected gradient descent algorithm. They also show that their implementation of the CCNN algorithm on several versions of the MNIST dataset and the CIFAR-10 dataset are able to obtain results comparable to state-of-the-art models from the nonconvex class of models.

In this thesis, we have explained the mathematical details for how the CCNN model class is constructed and described the algorithm that can be used to train such a model. The algorithm makes use of several steps. In one step a kernel matrix is approximated to find new features via $\mathbf{K} \approx \mathbf{Q}\mathbf{Q}^\top$ with $\mathbf{Q}$ being a representation of the original patches $\mathbf{Z}$, and the features used for the projected gradient descent in the CCNN algorithm. For this approximation step, a Nystrom dimension m must be chosen which influences the accuracy of the approximation step. In the next step of the algorithm, the coefficients of the model are projected on the nuclear norm ball for some chosen regularization parameter r. Much like in techniques such as ridge and lasso regression, the size of this parameter indicates the 'size' of the parameter space and therefore influences the results.

The influence of these two hyperparameters are discussed at best marginally in [1]. Furthermore, the CCNN algorithm was applied only on image data, which is the traditional application for convolutional neural networks. In this work, we have presented the application of the CCNN on two simulated datasets to show the influence of the hyperparameters m and r. Furthermore, we have shown that the CCNN also works on the task of classifying text data, where in previous work [17] this has only been done by a CNN. Also, the implementation of the CCNN by [1], while publicly available, is not ready for implementation on datasets other than specifically those used in [1] and in specifically the manner described by them. Their scripts were written specifically for the MNIST and CIFAR-10 data sets with no easy way to change any variables to apply it to other data. For this thesis, the entire algorithm has been written from the ground up in a structurally organized manner so that hyperparameters for the application of it to unstructured and text data can be more efficiently selected and the training patterns more easily visualized.

The question of whether the CCNN algorithm can be successfully applied on text data can be answered positively. The CCNN algorithm achieves an acceptable performance of around 90% classification accuracy on the *Clickbait* data set, which however is still lower than performance achieved by a simple CNN.

The question of how the m and r hyperparameters influence the training paths has also been answered. On the simulated data sets, we have shown that generally, the loss function is easier to minimize for higher values of m and r, although it takes some more iterations before the validation accuracy catches up with the smaller models. In the *clickbait* data set, the results show that best results are achieved for the bigger models, since the extra information from a higher m and/or larger nuclear norm allowance r lets the model find the best parameters.

Comparing the advantages and disadvantages of the CCNN and the CNN models, we found that in general the simple CNNs implemented in this thesis were able to achieve similar or better results than the CCNN. The CCNNs however, were faster to train. This point is especially impressive considering the fact that the CNNs were implemented using the *TensorFlow/Keras* libraries in Python, which make the back-propagation calculations significantly faster by allowing them to be computed on the computer's Graphical Processing Unit. The CCNN algorithm, which was coded using *NumPy* and *numexpr*, could quickly achieve their optimal solution by comparison. Another point that would count in favour of the CCNN in this regard is the fact that the CNN modules have been extensively studied and optimized, while the CCNN algorithm is not yet fully optimized.

The CCNN algorithm has an inherent drawback which could severely limit its application. It is the need to compute and approximate the kernel matrix $\mathbf{K}$. For meaningful problems, this kernel matrix will be very large. In fact, the applications by [1] required a 10GB and 50GB memory requirement for the MNIST and CIFAR10 data sets respectively, which is well beyond the average computer's capabilities nowadays. CNNs modeling these data sets using the aforementioned *Tensorflow/Keras* libraries are much more achievable by comparison. Another implication of the use of this kernel seems to be that it is especially useful for data sets where the input and output are related to each other in an unidirectional manner, as per a function $f : X \mapsto Y$. It is not immediately clear how other deep neural network architectures with a feedback loop of information, such as an RNN or a LSTM, could be meaningfully convexified in this manner.

The last point that can be made is that of whether convexity of the optimization problem is important. There are certainly reasons to argue why it is not. In many modern-day applications of deep learning models, the main point of interest is to understand how the non-convex algorithms work and how to make them perform better. When faced with a host of non-convex models that give a great flexibility in how to approach a problem, it can be difficult to motivate having to pay a price in numerical complexity for wanting to insist on convexity. Modern machine learning tasks, such as vision-speech and language recognition tasks are currently implemented with deep hierarchical models. It is precisely because these problems are inherently non-linear that such a hierarchical representation is necessary. In these cases, it can be argued that the problem that we are trying to model is inherently non-convex. On the other hand, it could also be argued that the convex methods to tackle these problems have simply not been sufficiently developed yet. In any case, this work has shown some applications of shallow convexified convolutional neural networks, but any formal study of convex relaxation of deep neural networks is still an open problem.

A Appendix

A.1 Theorems

Theorem 1 (Nonparametric Representer Theorem)**.** *(by [10]) Suppose we are given a nonempty set $\mathcal{X}$, a positive definite real-valued kernel k on $\mathcal{X} \times \mathcal{X}$, a training sample $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n) \in \mathcal{X} \times \mathbb{R}$, a strictly monotonically increasing real-valued function g on $[0, \infty]$, an arbitrary cost function $c : (\mathcal{X} \times \mathbb{R}^2)^m \to \mathbb{R} \cup \{\infty\}$, and a class of functions*

$$\mathcal{F}\Big\{ f \in \mathbb{R}^{\mathcal{X}} \big| f(\cdot) = \sum_{i=1}^{\infty} \beta_i k(\cdot, \mathbf{x}_i), \beta_i \in \mathbb{R}, \mathbf{x}_i \in \mathcal{X}, \|f\| < \infty \Big\}. \tag{57}$$

Here, $\|\cdot\|$ is the norm in the RKHS $\mathcal{H}_k$ associated with k. Then any $f \in \mathcal{F}$ minimizing the regularized risk functional

$$c((\mathbf{x}_1, y_1, f(\mathbf{x}_1)), \ldots, (\mathbf{x}_n, y_n, f(\mathbf{x}_n))) + g(\|f\|) \tag{58}$$

admits a representation of the form

$$f(\cdot) = \sum_{i=1}^{n} \alpha_i k(\cdot, \mathbf{x}_i). \tag{59}$$

Note how in Theorem 1 above, the functions $f \in \mathbb{R}^{\mathcal{X}}$ admit an infinite-dimensional form, yet the function which minimizes the cost function is finite dimensional, with m being the sample size.

A.2 Minor proofs

Lemma 12 (Trace of a concatenated matrix)**.** *When any number of matrices $\mathbf{M}_i$, $i \in [k]$ are concatenated horizontally such that $\mathbf{M} = [\mathbf{M}_1, \ldots, \mathbf{M}_k]$, then*

$$tr(\mathbf{M}^T\mathbf{M}) = \sum_{i}^{k} tr(\mathbf{M}_i^T\mathbf{M}_i).$$

Proof. The matrix multiplication $\mathbf{M}^{\mathrm{T}}\mathbf{M}$ will result in a square matrix that can be partitioned into blocks such that $(\mathbf{M}^{\mathrm{T}}\mathbf{M})_{ij} = \mathbf{M}_i^{\mathrm{T}}\mathbf{M}_j$. Because $\mathbf{M}_i^{\mathrm{T}}\mathbf{M}_j$ will lie along the trace of $(\mathbf{M}^{\mathrm{T}}\mathbf{M})$ $\forall i = j$, we can see that the original claim holds. □

A.3 Definitions

Definition 5. (Karush–Kuhn–Tucker conditions). Given are a function $f(\mathbf{w})$, which needs to be minimized given some inequality constraint functions $g_i(\mathbf{w})$ and equality constraint functions $h_j(\mathbf{w})$, and all are differentiable at point $\hat{\mathbf{w}}$. Then, if $\hat{\mathbf{w}}$ is a local optimum, then there exist constants λ_i and θ_i, such that

$$\nabla f(\hat{\mathbf{w}}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\hat{\mathbf{w}}) + \sum_{j=1}^{l} \theta_j \nabla h_j(\hat{\mathbf{w}}) = 0,$$

as long as the following conditions hold:

1. $g_i(\hat{\mathbf{w}}) \leq 0, \forall i$,
2. $h_j(\hat{\mathbf{w}}) = 0, \forall j$,
3. $\lambda_i \geq 0, \forall i$,
4. $\lambda_i g_i(\hat{\mathbf{w}}) = 0, \forall i$.

A.4 Python code: Network algorithms

A.4.1 CCNN Algorithm

```
""" This module will contains the functions required for:

    1) Helper functions
    2) Projecting a matrix on nuclear norm ball
    3) Projected Gradient Descent of a matrix on the nuclear norm ball
    4) train CCNN on radial data and store results on disk
    5) cnstruct loss, accuracy plots for the CCNN on radial training
    6) construct loss, accuracy plots for the CCNN on sigmoid training
"""

# public libraries
import numpy as np
from numpy import linalg as LA
import sys
import numexpr as ne
from sklearn.preprocessing import label_binarize
import time
import pickle as pkl

##########################################
##### 1) Helper functions ##############
##########################################
# helper functions
def tprint(s):
    """ Enhanced print function with time added to the output.
    Source: Zhang et al.
    """
    tm_str = time.strftime("%H:%M:%S", time.gmtime(time.time()))
    print(tm_str + ": " + str(s))
    sys.stdout.flush()

##########################################
### 2) ALGORITHM 2: project a matrix on nuclear norm ball
##########################################
""" Algorithm 2 in MSc. Thesis of Maarten van Schaik
The code for these three functions is heavily inspired by that of Zhang et al.
in their CCNN script.

The three functions below compute the steps for Algorithm 2.

Main function: project_to_trace_norm, which requires the functions
euclidean_proj_simplex, euclidean_proj_l1ball
"""

def project_to_nuclear_norm(A, R, P, nystrom_dim, d2):
    """ Main function for Algorithm 2
    Dependencies: euclidean_proj_simplex, euclidean_proj_l1ball

    Parameters
    ----------
```

```
    A: numpy array,
        matrix to be projected onto the nuclear norm ball
    R: int,
       upper bound of nuclear norm.
    P: int,
       number of patches (on clickbait: sequence length)
    nystroem_dim: int,
                  Nystroem dimension. In Thesis: m
    d2: int,
        Number of classes for the categorical classification

    Returns
    -------
    Ahat: numpy array,
          Projection of A on the nuclear norm ||A||_{*} = R
    U, s, V: singular vectors and values of A
    """
    A = A.reshape((d2-1)*P, nystrom_dim)
    # A = np.reshape(A, ((n_classes-1)*P, nystroem_dim))
    (U, s, V) = LA.svd(A, full_matrices=False)
    s = euclidean_proj_l1ball(s, s=R)
    Ahat = np.reshape(np.dot(U, np.dot(np.diag(s), V)), ((d2-1), P*nystrom_dim))
    return Ahat, U, s, V

def euclidean_proj_simplex(v, s=1):
    """ Compute the Euclidean projection on a positive simplex
    Solves the optimisation problem (using the algorithm from [1]):
        $min_w 0.5 * || w - v ||_2^2 , s.t. \sum_i w_i = s, w_i >= 0$
    Parameters
    ----------
    v: (n,) numpy array,
       n-dimensional vector to project
    s: int, optional, default: 1,
       radius of the simplex
    Returns
    -------
    w: (n,) numpy array,
       Euclidean projection of v on the simplex
    Notes
    -----
    The complexity of this algorithm is in O(n log(n)) as it involves sorting v.
    Better alternatives exist for high-dimensional sparse vectors (cf. [1])
    However, this implementation still easily scales to millions of dimensions.
    References
    ----------
    [1] Efficient Projections onto the .1-Ball for Learning in High Dimensions
        John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra.
        International Conference on Machine Learning (ICML 2008)
        http://www.cs.berkeley.edu/~jduchi/projects/DuchiSiShCh08.pdf
    """
    assert s > 0, "Radius s must be strictly positive (%d <= 0)" % s
    n, = v.shape # will raise ValueError if v is not 1-D
    # check if we are already on the simplex
    if v.sum() == s and np.alltrue(v >= 0):
```

```
        # best projection: itself!
        return v
    # get the array of cumulative sums of a sorted (decreasing) copy of v
    u = np.sort(v)[::-1]
    cssv = np.cumsum(u)
    # get the number of > 0 components of the optimal solution
    rho = np.nonzero(u * np.arange(1, n+1) > (cssv - s))[0][-1]
    # compute the Lagrange multiplier associated to the simplex constraint
    theta = (cssv[rho] - s) / (rho + 1.0)
    # compute the projection by thresholding v using theta
    w = (v - theta).clip(min=0)
    return w

def euclidean_proj_l1ball(v, s=1):
    """ Compute the Euclidean projection on a L1-ball
    Solves the optimisation problem (using the algorithm from [1]):
        $min_w 0.5 * || w - v ||_2^2 , s.t. || w ||_1 <= s$
    Parameters
    ----------
    v: (n,) numpy array,
       n-dimensional vector to project
    s: int, optional, default: 1,
       radius of the L1-ball
    Returns
    -------
    w: (n,) numpy array,
       Euclidean projection of v on the L1-ball of radius s
    Notes
    -----
    Solves the problem by a reduction to the positive simplex case
    See also
    --------
    euclidean_proj_simplex
    """
    assert s > 0, "Radius s must be strictly positive (%d <= 0)" % s
    n, = v.shape # will raise ValueError if v is not 1-D
    # compute the vector of absolute values
    u = np.abs(v)
    # check if v is already a solution
    if u.sum() <= s:
        # L1-norm is <= s
        return v
    # v is not already a solution: optimum lies on the boundary (norm == s)
    # project *u* on the simplex
    w = euclidean_proj_simplex(u, s=s)
    # compute the solution to the original problem on v
    w *= np.sign(v)
    return w

############################################
# 3) ALGORITHM 3: Projected Gradient Descent of a matrix on the nuclear norm
    ↪ ball
############################################
def evaluate_classifier(X_train, X_test, Y_train, Y_test, A, d2):
```

```
    """ Evaluates quality of the classifier by computing the loss and the
    classification error in the train and test set. The loss used is categorical
    cross-entropy and the classification error is zero-one loss.
    """
    n_train = X_train.shape[0]
    n_test = X_test.shape[0]
    eXAY = np.exp(np.sum((np.dot(X_train, A.T)) * Y_train[:,0:(d2-1)],
                         axis=1))
    eXA_sum = np.sum(np.exp(np.dot(X_train, A.T)), axis=1) + 1
    loss = - np.average(np.log(eXAY/eXA_sum))

    predict_train = np.concatenate((np.dot(X_train, A.T), np.zeros((n_train, 1),
                                                dtype=np.float32)), axis=1)
    predict_test = np.concatenate((np.dot(X_test, A.T), np.zeros((n_test, 1),
                                               dtype=np.float32)), axis=1)

    error_train = np.average(np.argmax(predict_train, axis=1) != \
                             Y_train.reshape(n_train,))
    error_test = np.average(np.argmax(predict_test, axis=1) != \
                            Y_test.reshape(n_test,))

    return loss, error_train, error_test

def projected_gradient_descent(X_train,
                               Y_train,
                               X_test,
                               Y_test,
                               R,
                               n_iter,
                               learning_rate,
                               print_iterations):
    """ In order to solve optimization problem in Algorithm 1, projected
    gradient descent is used. At iteration t, using a step size $\eta > 0$,
    it forms the new matrix $A^{t+1}$ based on the previous iterate $A^t$
    according to:

        $$A^{t+1} = \Prod_R(A^t - \eta \nabla_A L(A^t)),$$

    where $\nabla_A L(A^t)$ denotes the gradient of the objective function
    defined in Algorithm 1,, and $\Prod_R$ denotes the Euclidean projection
    onto the nuclear norm ball $\{A: \|A\|_{*} \leq R\}$.

    ------------------------------------------------------------------------
        ↪

    This function takes as input Will return the learned
    parameter matrix A \in R^{m , P*d_2}.

    X_train, X_test: train and test data (X_reduced[0:n_train] etc) Respective
                     sizes decided by n_train. Here is $Q \in R^{n, P, m}$,
                     the low-dimensional feature representation of the
                     kernelized patches (as returned by the Nystrom transform)

    Y_train, Y_test: train and test labels.
```

```
For clarification:
P: Number of patches. In the clickbait data, the 20-token sequences are
    evaluated with stride=1, (so each token is its own patch), so P=20.

nystrom_dim: Dimension of the $Q$ matrix. In clickbait data, there is 1
                channel and nystrom_dim=m, such that nystrom_dim=1*m=m.

R: toplevel input by user for. It is the hyperparameter R: the radius of
      the nuclear norm ball onto which the parameter matrix A is projected;
"""
d2 = len(np.unique(Y_train))
n_train, P, nystrom_dim = X_train.shape
n_test = X_test.shape[0]
X_train = X_train.reshape(n_train, P*nystrom_dim)
X_test = X_test.reshape(n_test, P*nystrom_dim)
A = np.random.randn((d2-1), P*nystrom_dim)
A_sum = np.zeros(((d2-1), P*nystrom_dim), dtype=np.float32)

# setup for objects to store performance during training
loss_history = np.array(())
error_train_history = np.array(())
error_test_history = np.array(())

# Projected Stochastic Gradient Descent
mini_batch_size = 50
nr_of_mini_batches = 10
for t in range(n_iter):
    # Non-projected Stochastic Gradient Descent
    for i in range(0, nr_of_mini_batches):
        """ This double for loop is largely inspired by Zhang et al's
        version in their CCNN code.

        For each batch in nr_of_mini_batches, randomy select mini_batch_size
        training samples inputs and labels. For this mini_batch_size samples,
        calculate the steps for stochastic gradient descent such that:

        $\nabla_A{_k} L(A_k) =
              frac{1}{n}X(frac{exp(XA_kˆT)}{\sum_{k=1}ˆK exp(XA_kˆT)}
                  ↪ - Y)$

        where A_k are the parameters for class K, X the mini batch of data,
        Y the targets of the mini batch, and n the mini batch size. L(A_k)
        is the loss function for multiclass classification (softmax)

        The update rule is then that $A_{t+1} =
                                          A_{t} - \gamma \nabla_{A_k}L(A_k)
                                              ↪ $

        After the update is applied, the new coefficients are projected on
        the nuclear norm.
        """
        # randomly sample mini_batch_size (=50) patches
        index = np.random.randint(0, n_train, mini_batch_size)
```

```
        X_sample = X_train[index] # dimensions: (50, P*nystrom_dim)
        # one column removed (because inferred from other columns):
        Y_sample = Y_train[index, 0:(d2-1)]

        # stochastic gradient descent
        XA = np.dot(X_sample, A.T)
        eXA = ne.evaluate("exp(XA)")
        eXA_sum = np.sum(eXA, axis=1).reshape((mini_batch_size, 1)) + 1
        diff = ne.evaluate("eXA/eXA_sum - Y_sample")
        grad_A = np.dot(diff.T, X_sample) / mini_batch_size
        A -= learning_rate * grad_A # average A after nr_of_mini_batches
            ↪ times

    # projection to nuclear norm
    A, U, s, V = project_to_nuclear_norm(A=A,
                                         R=R,
                                         P=P,
                                         nystrom_dim=nystrom_dim,
                                         d2=d2)
    A_sum += A
    if (t+1) % print_iterations == 0:
        # percentage of 'variance' in top 25 singular values:
        dim = np.sum(s[0:25]) / np.sum(s)
        A_avg = A_sum / 250
        loss, error_train, error_test = evaluate_classifier(X_train=X_train,
                                                            X_test=X_test,
                                                            Y_train=Y_train,
                                                            Y_test=Y_test,
                                                            A=A_avg,
                                                            d2=d2)
        loss_history = np.append(loss_history, np.array(loss))
        error_train_history = np.append(error_train_history,
                                        np.array(error_train))
        error_test_history = np.append(error_test_history,
                                       np.array(error_test))

        A_sum = np.zeros(((d2-1), P*nystrom_dim), dtype=np.float32) # reset
            ↪  A_sum

        tprint("iter " + str(t+1) +
               ": loss=" + str(loss) +
               ", train accuracy =" + str(error_train) +
               ", test accuracy =" + str(error_test))

    history = np.concatenate((loss_history[:,np.newaxis],
                              error_train_history[:,np.newaxis],
                              error_test_history[:,np.newaxis]),
        axis=1)

""" Once the final iterations have been made, the final trace-projected
coefficients are calculated and returned.
"""
A_avg, U, s, V = project_to_nuclear_norm(A=np.reshape(A_avg, ((d2-1)*P,
                                                              nystrom_dim)),
```

```
                                        R=R,
                                        P=P,
                                        nystrom_dim=nystrom_dim,
                                        d2=d2)
    dim = min(np.sum((s > 0).astype(int)), 25)
    return A_avg, V[0:dim], history

##########################################
## ALGORITHM 1: LEARN A CCNN ##########
##########################################
class ccnn:
    """ Function class to compute Algorithm 1 from MSc. Thesis.

    Functions:
        __init__: initializes the ccnn class.
        construct_Q: From the input data, construct Q by approximating the
                     kernel matrix K.
        train: Trains the CCNN on Q and Y.
    """
    def __init__(self,
                 input_file,
                 label_file,
                 n_train,
                 nystrom_dim,
                 gamma,
                 R,
                 learning_rate,
                 n_iter,
                 print_iterations):
        """ Initializes the CCNN model with input from user.

        Parameters
        ----------
        input_file, label_file: path directories for X and Y.
                                X must be a (N, P, d1) array.
                                Y must be a (N,) array.
        nystrom_dim: Nystroem dimension used to approximate Q during
                     step 1 of Algorithm 1 in MSc. Thesis. In Thesis: "m".
        gamma: hyperparameter for the RBF kernel.
        R: Nuclear Norm radius to project A on: ||A||_{*} = R
        n_iter: number of iterations for the Projected Stochastic Gradient Descent
        print_iterations: how often to print current loss and accuracy to
                          the console. 1 means it prints each iteration.
        """
        tprint("read from " + input_file)
        # Storing data
        self.X_raw = pkl.load(open(input_file, "rb"))
        self.label = pkl.load(open(label_file, "rb"))[:, 0]
        # Storing data properties
        self.d2 = np.unique(self.label).shape[0]
        self.n = self.X_raw.shape[0]
        self.P = self.X_raw.shape[1]
        self.d1 = self.X_raw.shape[2]
        # Storing hyperparameters
```

```
        self.n_train = n_train
        self.n_test = self.n - self.n_train
        self.nystrom_dim = nystrom_dim # in Thesis: m
        self.gamma = gamma
        self.R = R
        self.learning_rate = learning_rate
        self.n_iter = n_iter
        self.print_iterations = print_iterations

        # construct patches
        # This is an artifact from when X_raw was not neccesarily supplied as a
        # 3-dimensional array, and n, P, d1 were given as user input.
        # For sequence data it's easy: just use .reshape
        self.Z = self.X_raw.reshape(self.n,
                                    self.P,
                                    self.d1)
        tprint("Data contains " + str(self.n) + " samples, with " +
               str(self.P) + " patches of dimension " + str(self.d1) + ".")
        tprint("Output contains " + str(self.n) + " samples, with " +
               str(self.d2) + " classes.")

    def construct_Q(self, feature_normalization=True):
        """ Step 1 and 2 in Algorithm 1 in MSc. Thesis.
        Computes Q, such that QQ^T \approx K, where K is the RBF kernel
            ↪ matrix.
        Also applies normalization to the features by default. This is carried
        over by example from the CCNN code of Zhang et. al

        Input
        Z_train, Z_test: (N,P,d1) arrays. Each Z[i,:,:] is one Z(x_i).
                         Result from __init__.

        Output
        Q_train, Q_test: (N,P,m) arrays Each Q[i,:,:] is one Q(x_i).
                         Used in train() function below.
        """
        from sklearn.kernel_approximation import Nystroem
        import numpy as np
        import math
        tprint("Using Scikitlearn Nystroem function")
        tprint("Creating Q...")
        Z_train = self.Z[0:self.n_train].reshape((self.n_train*self.P, self.d1))
        Z_test = self.Z[self.n_train:self.n].reshape((self.n_test*self.P, self.d1))
        transformer = Nystroem(gamma=self.gamma, n_components=self.
            ↪ nystrom_dim)
        transformer = transformer.fit(X=Z_train)
        Q_train = transformer.transform(Z_train)
        Q_test = transformer.transform(Z_test)
        self.Q_train = Q_train.reshape((self.n_train, self.P, self.nystrom_dim))
        self.Q_test = Q_test.reshape((self.n_test, self.P, self.nystrom_dim))

        if feature_normalization==True:
            self.Q_train = self.Q_train.reshape((self.n_train*self.P,
                                                 self.nystrom_dim))
```

```
            self.Q_train -= np.mean(self.Q_train, axis=0)
            self.Q_train /= LA.norm(self.Q_train) / math.sqrt(self.n_train*self.P)
            self.Q_train = self.Q_train.reshape((self.n_train,
                                                 self.P, self.nystrom_dim))
            self.Q_test = self.Q_test.reshape((self.n_test*self.P,
                                               self.nystrom_dim))
            self.Q_test -= np.mean(self.Q_test, axis=0)
            self.Q_test /= LA.norm(self.Q_test) / math.sqrt(self.n_train*self.P)
            self.Q_test = self.Q_test.reshape((self.n_test,
                                               self.P, self.nystrom_dim))
    # Training CCNN
    def train(self):
        """ Algorithm 1 from MSc. Thesis.
        Trains the CCNN using Projected Stochastic Gradient Descent.

        It solves the constrained optimization problem of step 3 in Algorithm 1.

        Requires:
        -----
        Q_train, Q_test: Output from construct_Q function
        Y_train, Y_test: User input on class level

        Label_binarize will create a one-hot encoding matrix with K-1 columns,
        showing a 1 in each row only once to indicate which class the case
        belongs to.

        Parameters
        ----------
        n_iter: number of iterations for the Projected Stochastic Gradient Descent
        print_iterations: how often to print current loss and accuracy to
                          the console. 1 means it prints each iteration.
        """
        tprint("Training CCNN using projected stochastic gradient descent...")
        from sklearn.preprocessing import label_binarize
        binary_label = label_binarize(self.label, classes=range(0, self.d2))

        self.Y_train=binary_label[0:self.n_train]
        self.Y_test=binary_label[self.n_train:]

        self.A, self.filter, self.train_history = \
            projected_gradient_descent(X_train=self.Q_train,
                                       Y_train=self.Y_train,
                                       X_test=self.Q_test,
                                       Y_test=self.Y_test,
                                       n_iter=self.n_iter,
                                       print_iterations=self.print_iterations,
                                       R=self.R,
                                       learning_rate=self.learning_rate)
```

A.4.2 CNN Algorithm

```
""" This module contains functions to:
    1) Set up a simple CNN network to be applied on the Clickbait dataset
```

```
"""
# public libraries
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalMaxPooling1D
import time, sys
########################################
# 1) Set up a Fully-connected neural network (non-convexified!)
########################################
class ConvNeuralNet_binary:
    def __init__(self):
        pass
    """ Instantiates a CNN object in Python.
    setup_model: Set up a simple 1D-convolutional neural network. It uses
        ↪  the
                embedded sentences as input then runs them through a
                    ↪ number of
                filters, which are summarized using global max pooling. A
                2-node and softmax layer combination is used to calculate
                    ↪ the
                output.

    fit_model: Starts training the neural network created using
        ↪ setup_model.
    """
    def setup_model(self,
                    embedding_matrix,
                    max_sequence_length,
                    n_filters,
                    k_size):
        # Save important parameters into the class
        self.max_sequence_length = max_sequence_length
        self.n_filters = n_filters
        self.k_size = k_size
        self.vocab_size, self.vocab_dim = embedding_matrix.shape

        # Set up the simple 1D-Convolutional Neural Network
        tprint("Building a model...")
        embedding_layer = Embedding(self.vocab_size,
                                    self.vocab_dim,
                                    weights=[embedding_matrix],
                                    input_length=self.max_sequence_length,
                                    trainable=False,
                                    name="Embedding")
        self.model = Sequential()
        self.model.add(embedding_layer)
        self.model.add(Conv1D(filters=self.n_filters,
                              kernel_size=self.k_size,
                              activation='relu',
                              strides=1,
                              padding='valid',
                              name="Conv1D"))
        self.model.add(GlobalMaxPooling1D(name="Max_Pooling"))
```

```
        self.model.add(Dense(2, activation='softmax', name="Dense_Softmax
            ↪ "))
        self.model.compile(loss='categorical_crossentropy',
                           optimizer='rmsprop',
                           metrics=['accuracy'])

    def fit_model(self,
                  X_train, X_test, Y_train, Y_test,
                  batch_size=10, epochs=128):
        self.batch_size = batch_size
        self.epochs = epochs
        tprint("Fitting 1D-Convolutional Neural Net with "+str(self.
            ↪ n_filters)+
               " filters in the hidden layer, for "+str(self.epochs)+"
                   ↪ epochs...")
        self.history = self.model.fit(X_train, Y_train,
                                      epochs=self.epochs,
                                      batch_size=self.batch_size,
                                      validation_data=(X_test, Y_test),
                                      verbose=1)
```

A.4.3 NN Algorithm

```
""" This module contains functions to:
    1) Helper functions
    2) Set up a fully connected neural network class
"""

# public libraries
import numpy as np
from keras.models import Sequential
from keras.layers import Dense

########################################
# 1) Helper functions
########################################
def TestError(Y, Yhat):
    """ Used in NEuralNet_binary.fit_model()
    Calculates the prediction error
    """
    pred_error = (Y - Yhat)**2
    return(np.average(pred_error))
########################################
# 2) Set up a Fully-connected neural network (non-convexified!)
########################################
# classes
class NeuralNet_binary:
    """
    Set up to fit a very simple fully-connected neural network suited for
    classification tasks. Because of the classification tasks it performs
        ↪ , it
    will train binary cross-entropy as loss function, measure accuracy by
        ↪  zero-
```

```
    one loss and optimize using adam.
    """
    def __init__(self):
        pass

    def setup_model(self, nr_hidden, input_dim):
        # create model
        self.nr_hidden = nr_hidden
        self.model = Sequential()
        self.model.add(Dense(self.nr_hidden,
                             input_dim=input_dim,
                             activation='relu'))
        self.model.add(Dense(1, activation='sigmoid'))
        self.model.summary()
        # Compile model
        self.model.compile(loss='binary_crossentropy',
                           optimizer='adam',
                           metrics=['accuracy'])

    def fit_model(self,
                  X_train, X_test, Y_train, Y_test,
                  batch_size=10, epochs=128):
        self.batch_size = batch_size
        self.epochs = epochs
        tprint("Fitting fully-connected Neural Net with "+str(self.
            ↪ nr_hidden)+
               " hidden nodes, for "+str(self.epochs)+"epochs...")
        self.history = self.model.fit(X_train, Y_train,
                                      epochs=self.epochs,
                                      batch_size=self.batch_size,
                                      validation_data=(X_test, Y_test),
                                      verbose=1)

        self.Yhat = self.model.predict(X_test)
        self.test_error = TestError(Y=Y_test, Yhat=self.Yhat)
```

A.5 Python code: data access

A.5.1 Clickbait data

```
"""
This module specifies a number of functions related to loading the
    ↪ clickbait
dataset and preparing it for use in (convexified and/or convolutional)
    ↪ neural
networks.

load_data: Loads the clickbait dataset (input and labels) and splits into
          train and test sets.

tokenize: Removes punctuation and tokenizes the rest of input data using
    ↪ the
```

```
        Keras tokenizer, then pads each input to the max length allowed
            ↪ .

prepare_embedding: Will prepare a matrix containing the token embeddings
    ↪ from
                the GloVe dataset. The user can choose between
                    ↪ 50/100/200/300
                dimensional token embeddings. Please note, you will
                    ↪ need
                the (large) GloVe files on your disk.

vectorize_tokens: This will create vectorized dataset by taking the
    ↪ tokenized
                inputs and using the embedding matrix created by
                prepare_embedding to fetch the required vectors.
"""

data_path = 'raw_clickbait_data/'

def load_data(TEST_SPLIT=0.2):
    """This function does the following:
    Load the clickbait datasets
    Select a holdout sample (input: HOLDOUT_SPLIT)
    returns: x_train, y_train, x_test, y_test
    """
    import numpy as np

    # Read in the raw data
    f = open(data_path+'clickbait_data', encoding="utf8")
    baitlines = [line.rstrip().lower() for line in f if len(line.rstrip()
        ↪ ) > 0]
    f.close()
    f = open(data_path+'non_clickbait_data', encoding="utf8")
    nobaitlines = [ line.rstrip().lower() for line in f if len(line.
        ↪ rstrip()) > 0]
    f.close()
    headlines = baitlines + nobaitlines
    longest = len(max(headlines, key=len)) # longest headline length
    print("Reading data files...")
    print("Parsed %d bait headlines." % (len(baitlines)))
    print("Parsed %d non bait headlines." % (len(nobaitlines)))
    print("Parsed %d headlines in total." % (len(headlines)))
    print("Longest headline is %d tokens before pre-processing" % longest
        ↪ )
    print("(so including punctuation etc).")

    labels = []
    for i in np.arange(len(baitlines)):
        labels.append(1)
    for i in np.arange(len(nobaitlines)):
        labels.append(0)

    labels_index = {'clickbait': 1, 'news': 0}
```

```
    # get a test set
    from random import shuffle
    # Given list1 and list2
    headlines_shuf = []
    labels_shuf = []
    indices = np.arange(len(headlines))
    shuffle(indices)
    for i in indices:
        headlines_shuf.append(headlines[i])
        labels_shuf.append(labels[i])
    nb_holdout_samples = int(TEST_SPLIT * len(headlines))
    x_train = headlines_shuf[:-nb_holdout_samples]
    y_train = labels_shuf[:-nb_holdout_samples]
    x_test = headlines_shuf[-nb_holdout_samples:]
    y_test = labels_shuf[-nb_holdout_samples:]
    print("Creating a holdout sample...")
    print("Took %d headlines as holdout sample." % nb_holdout_samples)
    print("Remaining %d headlines will be used for training." % len(
        ↪ x_train))

    return(x_train, y_train, x_test, y_test)

def tokenize(x_train, y_train, x_test, y_test,
             max_nb_words=400000,
             max_sequence_length=26):
    """This function must accomplish the following:
        Apply tokenization on headlines
        Apply tokenization on holdout headlines
        (input for both: MAX_NB_WORDS, MAX_SEQUENCE_LENGTH)
    """

    from keras.preprocessing.text import Tokenizer
    from keras.preprocessing.sequence import pad_sequences
    from keras.utils import np_utils
    import numpy as np

    print("Initializing Keras tokenizer, max. unique words allowed = %d"
        ↪ % max_nb_words)
    tokenizer = Tokenizer(num_words=max_nb_words)
    tokenizer.filters = '#$%&()*+,-./:;<=>@[\\]^_`{|}~\t\n'
    tokenizer.fit_on_texts(x_train)
    print("Applying tokenizer on training corpus...")
    x_train = tokenizer.texts_to_sequences(x_train)
    word_index = tokenizer.word_index
    print('Found %s unique tokens.' % len(word_index))

    print('Padding training sequences to max length %d' %
        ↪ max_sequence_length)
    x_train = pad_sequences(x_train, maxlen=max_sequence_length)

    y_train = np_utils.to_categorical(np.asarray(y_train))
    print('Shape of training x_train tensor:', x_train.shape)
    print('Shape of training y_train tensor:', y_train.shape)
```

```
    """
    Here starts a section dedicated to testing new headlines on which the
        ↪  model has
    not been trained and which may also contain words that are not in the
        ↪  vocabulary.
    These words will be removed from the new headline and a prediction
        ↪ will be made
    on the sentence as if it does not contain those words.
    """

    # latest test
    print("Applying tokenizer on holdout corpus...")
    x_test = tokenizer.texts_to_sequences(x_test)
    #word_index = tokenizer.word_index
    #print('Found %s unique tokens.' % len(word_index))
    print('Padding holdout sequences to max length %d' %
        ↪ max_sequence_length)
    x_test = pad_sequences(x_test, maxlen=max_sequence_length)
    y_test = np_utils.to_categorical(np.asarray(y_test))
    print('Shape of holdout x_test tensor:', x_test.shape)
    print('Shape of holdout y_test tensor:', y_test.shape)

    return(x_train, y_train, x_test, y_test, tokenizer)

def prepare_embedding(tokenizer, embedding_dim):
    """This function must accomplish the following:
        set up an embedding_index from pre-trained words (input:
            ↪ EMBEDDING_DIM)
        Set up embedding_matrix for our data
        """
    import numpy as np

    """
    Preparing the embedding layer by setting up an embeddings_index. This
        ↪  is a
    dictionary of 400000 pre-trained word vectors from GLOVE. These come
        ↪ in
    either 50, 100, 200 or 300 dimensions, depending on how precise you
        ↪ want to
    be and how much memory you are willing to use.
    """
    print("Initializing word vectors based on %d-dimensional GLOVE
        ↪ dictionary..." % embedding_dim)
    print("... (this could take a minute)")
    word_index = tokenizer.word_index

    embeddings_index = {}
    if embedding_dim == 50: f = open('raw_clickbait_data/glove.6B.50d.txt
        ↪ ', encoding="utf8")
    if embedding_dim == 100: f = open('raw_clickbait_data/glove.6B.100d.
        ↪ txt', encoding="utf8")
    if embedding_dim == 200: f = open('raw_clickbait_data/glove.6B.200d.
        ↪ txt', encoding="utf8")
```

```
    if embedding_dim == 300: f = open('raw_clickbait_data/glove.6B.300d.
        ↪ txt', encoding="utf8")
    for line in f:
        values = line.split()
        word = values[0]
        coefs = np.asarray(values[1:], dtype='float32')
        embeddings_index[word] = coefs
    f.close()

    """
    At this point we can leverage our embedding_index dictionary and our
        ↪ word_index
    to compute our embedding matrix:
    """

    count = 0
    for word, i in word_index.items():
        if word in embeddings_index:
            count += 1
    print("%d out of %d tokens present in GLOVE dictionary" % (count, len
        ↪ (word_index)))
    print("Words not found will be embedded as all-zeros.")

    embedding_matrix = np.zeros((len(word_index) + 1, embedding_dim))
    for word, i in word_index.items():
        embedding_vector = embeddings_index.get(word)
        if embedding_vector is not None:
            # words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector

    return(word_index, embedding_matrix)

def vectorize_tokens(data, embedding_matrix, embedding_dim=50,
    ↪ max_sequence_length=26):
    """
    This function will take as input tokenized data and return a
        ↪ dataframe in
    which the data has been vectorized.

    data: tokenized data that you want to vectorize
    embedding_matrix: matrix where the i'th row contains vector for token
        ↪  'i'
    embedding_dim: dimension per token vector
    max_sequence_length: max length of an individual input data

    clickbait example: tokenized data is provided as an array of shape (N
        ↪ , 26)
    due to truncation after 26 tokens. the 50-dimensional GloVe vectors
        ↪ are used
    for the embedding matrix. The return is an array of shape (N, 50*26).
    """
    import numpy as np

    i = -1
```

```
    n = len(data)
    data_vec = np.zeros((n, embedding_dim*max_sequence_length))
    for title in data:
        i += 1
        # vec = np.zeros(EMBEDDING_DIM*MAX_SEQUENCE_LENGTH)
        vec = np.array([])
        for token in title:
            token_vec = embedding_matrix[token]
            vec = np.append(vec, token_vec)
        data_vec[i] = vec
    return(data_vec)
```

A.5.2 Simulated data

```
""" This module contains functions to:
    1) Simulate sum-of-sigmoid model data
    2) Simulate radial model data
"""
# public libraries
import numpy as np
import pandas as pd


# helper functions
def tprint(s):
    """
    Enhanced print function with time added to the output.
    """
    import time, sys
    tm_str = time.strftime("%H:%M:%S", time.gmtime(time.time()))
    print(tm_str + ": " + str(s))
    sys.stdout.flush()

########################################
# 1) Simulate sum-of-sigmoid model data
########################################
# project-specific functions
def create_sigmoidsum_data(N, train_split,
                           signal_noise_ratio=2,
                           seed=1234,
                           verbose=True):
    """
    Simulates data using the sum-of-sigmoids structure, as described by
        ↪ [1].
    Sources:
        [1] Friedman J, Hastie T, Tibshirani R. The elements of
            ↪ statistical
            learning. New York: Springer series in statistics; 2001.
    """
    # helper functions
    def sigmoid(x):
        return(1/(1+np.exp(-x)))
    def sigmoid_y(X):
```

```
        a1 = np.array((3,3)).reshape(2,1)
        a2 = np.array((3,-3)).reshape(2,1)
        s1 = sigmoid(np.dot(a1.T, X.T))
        s2 = sigmoid(np.dot(a2.T, X.T))
        return(s1+s2)

    np.random.seed(seed=seed)
    p=2 # 2 dimensions hardcoded for now
    X = pd.DataFrame(np.random.rand(N, p))
    Y = X.apply(sigmoid_y, axis=1).loc[:,0] # apply sigmoid_y on all
        ↪ samples

    # signal-to-noise ratio:
    error_sigma = np.sqrt(np.var(Y))/np.sqrt(signal_noise_ratio)
    errors = np.random.randn(N) * error_sigma
    Y_obs = Y + errors
    Y_obs = np.array(Y_obs).reshape(N, 1)
    X = np.array(X)
    X_train = X[0:np.int(N*train_split), :]
    X_test = X[np.int(N*train_split):N, :]
    Y_train = Y_obs[0:np.int(N*train_split)]
    Y_test = Y_obs[np.int(N*train_split):N]
    truemean = np.mean(Y.reshape(N,1)[0:np.int(N*train_split)])
    Y_train_bin = (Y_train > truemean)*1
    Y_test_bin = (Y_test > truemean)*1

    if verbose==True:
        tprint("Simulated data: sum of sigmoids model")
        tprint("Simulated "+str(N)+" datapoints, of which "+\
               str(N*train_split)+" test points")
    return(X_train, X_test, Y_train, Y_test, Y_train_bin, Y_test_bin)

########################################
# 1) Simulate radial model data
########################################
def create_radial_data(N, p, train_split,
                       signal_noise_ratio=4,
                       seed=1234, verbose=True):
    """
    Simulates data using the radial function structure, as described by
        ↪ [1].
    Sources:
        [1] Friedman J, Hastie T, Tibshirani R. The elements of
            ↪ statistical
           learning. New York: Springer series in statistics; 2001.
    """
    def radial(x):
        t = x**2
        psi = np.sqrt((1/(2*np.pi))) * np.exp(-t/2)
        return(psi)

    def radial_y(X):
        return(np.prod(radial(X)))
```

```
    np.random.seed(seed)
    X = pd.DataFrame(np.random.rand(N, p))
    Y = X.apply(radial_y, axis=1)
    # signal-to-noise ratio:
    error_sigma = np.sqrt(np.var(Y))/np.sqrt(signal_noise_ratio)
    errors = np.random.randn(N) * error_sigma
    Y_obs = Y + errors
    Y_obs = Y_obs.reshape(N, 1)
    X = np.array(X)
    X_train = X[0:np.int(N*train_split), :]
    X_test = X[np.int(N*train_split):N, :]
    Y_train = Y_obs[0:np.int(N*train_split)]
    Y_test = Y_obs[np.int(N*train_split):N]
    truemean = np.mean(Y.reshape(N,1)[0:np.int(N*train_split)])
    Y_train_bin = (Y_train > truemean)*1
    Y_test_bin = (Y_test > truemean)*1

    if verbose==True:
        tprint("Simulated data: Radial function model")
        tprint("Simulated "+str(N)+" datapoints, of which "+\
               str(N*train_split)+" test points")
    return(X_train, X_test, Y_train, Y_test, Y_train_bin, Y_test_bin)
```

A.6 Python code: applications

A.6.1 CCNN on Clickbait

```
""" This module contains the functions required to:
    1) Load, tokenize and vectorize clickbait dataset and store on disk
    2) Train a CCNN on the vectorized clickbait data and store loss, acc.
        ↪  on disk
    3) Make loss and accuracy plots (not included in report. See GitHub)
"""
# project-specific modules
# Public libraries
import pickle as pkl
import numpy as np
# Load project-specific modules
import functions_vectorizeClickbait # used in 1)
import functions_CCNN # used in 2)

########################################
# 1) LOAD, TOKENIZE AND VECTORIZE CLICKBAIT DATASET AND STORE ON DISK
########################################
""" Uses functions from 'functions_vectorizeClickbait.py'.

We are restricting our vocabulary to the 40000 most observed tokens. Non-
observed tokens will be embedded as all zeroes. Each sequence is padded (
    ↪ or
shortened) to a max length of 20. We are using 50-dimensional pre-trained
    ↪  word
embeddings from GloVe. See functions_vectorizeClickbait.py for more
    ↪ details.
```

```
"""
MAX_NB_WORDS=40000
MAX_SEQUENCE_LENGTH=20
EMBEDDING_DIM=50

# Load the raw data:
X_train, Y_train, X_test, Y_test = functions_vectorizeClickbait.load_data
    ↪ ()

# Now tokenize the data and pad them to max length of 20 tokens. Also,
    ↪ return
# the tokenizer (a keras.proprocessing.text.Tokenizer object)
X_train, Y_train, X_test, Y_test, tokenizer = \
   functions_vectorizeClickbait.tokenize(X_train,
                            Y_train,
                            X_test,
                            Y_test,
                            max_nb_words=MAX_NB_WORDS,
                            max_sequence_length=MAX_SEQUENCE_LENGTH)

# construct an embedding matrix from the GloVe vectors and a word_index
    ↪ dict.
word_index, embedding_matrix = \
functions_vectorizeClickbait.prepare_embedding(tokenizer=tokenizer,
                                 embedding_dim=EMBEDDING_DIM)

# Leverage the embedding matrix to vectorize our data
X_train = functions_vectorizeClickbait.vectorize_tokens(X_train,
                                   embedding_matrix=embedding_matrix,
                                   embedding_dim=EMBEDDING_DIM,
                                   max_sequence_length=
                                       ↪ MAX_SEQUENCE_LENGTH)

X_train = X_train.reshape(X_train.shape[0], MAX_SEQUENCE_LENGTH,
    ↪ EMBEDDING_DIM)

X_test = functions_vectorizeClickbait.vectorize_tokens(X_test,
                                  embedding_matrix=embedding_matrix,
                                  embedding_dim=EMBEDDING_DIM,
                                  max_sequence_length=
                                      ↪ MAX_SEQUENCE_LENGTH)

X_test = X_test.reshape(X_test.shape[0], MAX_SEQUENCE_LENGTH,
    ↪ EMBEDDING_DIM)

# Save our vectorized data as a pkl object so we don't have to re-run
    ↪ code above
INPUT_FILE = 'data/clickbait_data.pkl'
LABEL_FILE = "data/clickbait_labels.pkl"
pkl.dump(np.vstack((X_test, X_train)), open(INPUT_FILE, "wb"))
pkl.dump(np.vstack((Y_test, Y_train)), open(LABEL_FILE, "wb"))

########################################
# 2) TRAIN CCNN ON CLICKBAIT DATA AND STORE RESULTS TO DISK
```

```
########################################
""" Uses functions from 'functions_CCNN.py'.

We are training 16 CCNNs on the clickbait data (which is constructed in
    ↪ step 1).
We train for the combinations for Nystroem dimensions (5, 25, 100, 200)
    ↪ and
nuclear norm bound for the parameter matrix ||A||_{*} = (.1, 1, 5, 100).

We use 750 training iterations for the Projected Stochastic Gradient
    ↪ Descent
step and half of the data (16000) is used for training. We use a learning
    ↪  rate
(eta) of 0.1 at each step and print results to console at each iteration.
"""


N_ITER = 750
N_TRAIN = 16000 # half of the total set
LEARNING_RATE = 0.1
PRINT_ITERATIONS = 1
INPUT_FILE = 'data/clickbait_data.pkl'
LABEL_FILE = "data/clickbait_labels.pkl"

loss_df = {}
train_acc = {}
test_acc = {}
for m in np.array((200, 100, 25, 5)):
    for r in np.array((100, 5, 1, 0.1)):
        model = functions_CCNN.ccnn(input_file=INPUT_FILE,
                        label_file=LABEL_FILE,
                        n_iter=N_ITER,
                        n_train=N_TRAIN,
                        learning_rate=LEARNING_RATE,
                        nystrom_dim=m,
                        R=r,
                        print_iterations=PRINT_ITERATIONS)
        model.construct_Q()
        model.train()
        params = "m"+str(m)+"r"+str(r)
        loss_df[params] = model.train_history[:,0]
        train_acc[params] = model.train_history[:,1]
        test_acc[params] = model.train_history[:,2]

# Phew, that took a while!

# Print validation accuracies for each combination
for key in test_acc:
    print(key, np.max(test_acc[key]))

# Save results for clickbait data
import pickle as pkl
pkl.dump(loss_df, open("results/clickbait_ccnn_loss.pkl", "wb"))
pkl.dump(train_acc, open("results/clickbait_ccnn_trainacc.pkl", "wb"))
```

```
pkl.dump(test_acc, open("results/clickbait_ccnn_testacc.pkl", "wb"))
```

A.6.2 CNN on Clickbait

```
""" This module contains functions to:
    1) Load clickbait data and create word embeddings
    2) Train CNN models on clickbait data

    For code to plot figures, see GitHub page

Notes:
We are using the module 'functions_vectorizeClickbait.py' to create the
    ↪ word-
embedding-version of the raw clickbait strings. See that script for more
details.
"""

# project-specific modules
import functions_vectorizeClickbait
import functions_CNN_clickbait
MAX_NB_WORDS=40000 # max nr of unique tokens in vocabulary
MAX_SEQUENCE_LENGTH=20 # Nr of tokens. In thesis: P
EMBEDDING_DIM=50 # GloVE embedding dimension.

########################################
# 1) Load clickbait data and create word embeddings
########################################
# Load the raw data:
X_train, Y_train, X_test, Y_test = \
functions_vectorizeClickbait.load_data(TEST_SPLIT=0.5)

# Now tokenize the data and pad them to max length of 20 tokens. Also,
    ↪ return
# the tokenizer (a keras.proprocessing.text.Tokenizer object)
X_train, Y_train, X_test, Y_test, tokenizer = \
functions_vectorizeClickbait.tokenize(X_train,
                      Y_train,
                      X_test,
                      Y_test,
                      max_nb_words=MAX_NB_WORDS,
                      max_sequence_length=MAX_SEQUENCE_LENGTH)

# construct an embedding matrix from the GloVe vectors and a word_index
    ↪ dict.
word_index, embedding_matrix = \
functions_vectorizeClickbait.prepare_embedding(tokenizer=tokenizer,
                                               embedding_dim=EMBEDDING_DIM)

########################################
# 2) Train CNN models and save loss, accuracy plots to disk
########################################
K_SIZE=1
N_ITERATIONS=40
```

```
for N_FILTERS in ((4, 16, 64)):
    model = functions_CNN_clickbait.ConvNeuralNet_binary()
    model.setup_model(embedding_matrix=embedding_matrix,
                      max_sequence_length=MAX_SEQUENCE_LENGTH,
                      n_filters=N_FILTERS,
                      k_size=K_SIZE)
    model.model.summary()
    history = model.fit_model(X_train, X_test,
                              Y_train, Y_test,
                              batch_size=10, epochs=N_ITERATIONS)
```

A.6.3 Clickbait CNN classifier function

```
def classify(line, model, tokenizer):
    """ Classify new examples into either clickbait or non-clickbait and
        ↪ return
        a verbose result.

        Line: a string
        model: a trained CNN model
        tokenizer: a tokenizer object created during step 1 in '
            ↪ clickbait_train_CNN.py'

        Examples
        --------
        # http://www.bbc.co.uk/bbcthree/article/2b2f79e8-c253-4d1b-9a87
            ↪ -44fe460e5b16
        classify(line="Confession booth: what your bikini waxer is really
            ↪  thinking",
                 model=model,
                 tokenizer=tokenizer)

        # https://www.bbc.co.uk/news/business-45055861
        classify(line="Carney: No-deal Brexit risk 'uncomfortably high'",
                 model=model,
                 tokenizer=tokenizer)

        # https://www.bbc.co.uk/news/business-45053528
        classify(line="Amazon tax bill falls despite profits leap",
                 model=model,
                 tokenizer=tokenizer)
    """
    import keras
    from keras.preprocessing.sequence import pad_sequences
    import numpy as np

    max_sequence_length = model.model.layers[0].input_length

    print("Analysing:", line)
    ex = line.lower()
    ex_words_seperated = keras.preprocessing.text.text_to_word_sequence(
        ↪ ex)
```

```
    ex_tokenized = np.array(tokenizer.texts_to_sequences(
        ↪ ex_words_seperated)).T
    #There is a hidden error when the word is not present in word_index.
        ↪ Therefore,
    #need to remove unknown words from new input headlines...
    if len(min(ex_tokenized)) == 0:
        removed = \
        np.asarray(ex_words_seperated)[np.where(np.logical_not(
            ↪ ex_tokenized).astype(int))]
        ex_tokenized = [sum(ex_tokenized, [])]
        print("Had to remove unknown word(s): %s" % removed)
    ex_input = pad_sequences(ex_tokenized, maxlen=max_sequence_length)
    pred = model.model.predict(ex_input)
    pred = np.argmax(pred) # 1 = clickbait, 0 = news
    if pred == 1:
        print("This line is clickbait!" '\n')
    if pred == 0:
        print("This isn't clickbait." '\n')
```

A.6.4 CCNN on simulated data

```
""" This module will contains the functions required for:

    1) simulate sum of sigmoids data and store to disk
    2) simulate radial model data and store to disk
    3) train CCNN on sigmoid data and store results to disk
    4) train CCNN on radial data and store results on disk

    For the code used to make the figures, see GitHub.
"""

########################################
# 1) SIMULATE SUM OF SIGMOIDS DATA AND STORE ON DISK
########################################
# project-specific modules
import functions_simulate_data
import pickle as pkl
import numpy as np
import matplotlib.pyplot as plt
""" SUM OF SIGMOIDS DATA"""
# Simulate data
N, TRAIN_SPLIT, SIGNAL_NOISE_RATIO = (10000, 0.5, 2)

(X_train, X_test,
 Y_train, Y_test,
 Y_train_bin, Y_test_bin) = functions_simulate_data.
     ↪ create_sigmoidsum_data(N=N,
                                                    train_split=
                                                        ↪ TRAIN_SPLIT,
                                                    signal_noise_ratio=
                                                        ↪ SIGNAL_NOISE_RATIO
                                                        ↪ ,
                                                    seed=1234)
```

```
X_train = np.expand_dims(X_train, axis=3)
X_test = np.expand_dims(X_test, axis=3)

X_total = np.vstack((X_train, X_test))
Y_total = np.vstack((Y_train_bin, Y_test_bin))

INPUT_FILE = 'data/SoS_X.pkl'
OUTPUT_FILE = "data/SoS.features"
LABEL_FILE = "data/SoS_Y_bin.pkl"
pkl.dump(X_total, open(INPUT_FILE, 'wb'))
pkl.dump(Y_total, open(LABEL_FILE, "wb"))

########################################
# 2) SIMULATE RADIAL DATA AND STORE ON DISK
########################################
import functions_simulate_data
import pickle as pkl
import numpy as np
""" SUM OF SIGMOIDS DATA"""
# Simulate data
N, TRAIN_SPLIT, SIGNAL_NOISE_RATIO = (10000, 0.5, 2)

(X_train, X_test,
 Y_train, Y_test,
 Y_train_bin, Y_test_bin) = functions_simulate_data.create_radial_data(N=
     ↪ N,
                      p=10,
                                                train_split=
                                                    ↪ TRAIN_SPLIT,
                                                signal_noise_ratio=
                                                    ↪ SIGNAL_NOISE_RATIO
                                                    ↪ ,
                                                seed=1234)
X_train = np.expand_dims(X_train, axis=3)
X_test = np.expand_dims(X_test, axis=3)

X_total = np.vstack((X_train, X_test))
Y_total = np.vstack((Y_train_bin, Y_test_bin))

INPUT_FILE = 'data/radial_X.pkl'
OUTPUT_FILE = "data/radial.features"
LABEL_FILE = "data/radial_Y_bin.pkl"
pkl.dump(X_total, open(INPUT_FILE, 'wb'))
pkl.dump(Y_total, open(LABEL_FILE, "wb"))


########################################
# TRAIN CCNN ON SIMULATED DATA AND STORE RESULTS TO DISK
########################################
import functions_CCNN
import pickle as pkl
import numpy as np
########################################
# 3) SUMS OF SIGMOID
```

```
#######################################
N_ITER = 200
N_TRAIN = 5000
LEARNING_RATE = 0.1
PRINT_ITERATIONS = 1

INPUT_FILE = 'data/SoS_X.pkl'
OUTPUT_FILE = "data/SoS.features"
LABEL_FILE = "data/SoS_Y_bin.pkl"

loss_df = {}
train_acc = {}
test_acc = {}
for m in np.array((25, 5, 2, 1)):
    for r in np.array((10, 1, 0.1, 0.01)):
        model = functions_CCNN.ccnn(input_file=INPUT_FILE,
                          output_file=OUTPUT_FILE,
                          label_file=LABEL_FILE,
                          n_iter=N_ITER,
                          n_train=N_TRAIN,
                          learning_rate=LEARNING_RATE,
                          nystrom_dim=m,
                          R=r,
                          print_iterations=PRINT_ITERATIONS)
        model.construct_Q()
        model.train()
        params = "m"+str(m)+"r"+str(r)
        loss_df[params] = model.train_history[:,0]
        train_acc[params] = model.train_history[:,1]
        test_acc[params] = model.train_history[:,2]

# Print validation accuracies for each combination
for key in test_acc:
    print(key, np.max(test_acc[key]))

# Save results for the radial data:
import pickle as pkl
pkl.dump(loss_df, open("results/SoS_ccnn_loss.pkl", "wb"))
pkl.dump(train_acc, open("results/SoS_ccnn_trainacc.pkl", "wb"))
pkl.dump(test_acc, open("results/SoS_ccnn_testacc.pkl", "wb"))

#######################################
# TRAIN CCNN ON SIMULATED DATA AND STORE RESULTS TO DISK
#######################################
#######################################
# 4) RADIAL
#######################################
import functions_CCNN
import pickle as pkl
import numpy as np

INPUT_FILE = 'data/radial_X.pkl'
OUTPUT_FILE = "data/radial.features"
LABEL_FILE = "data/radial_Y_bin.pkl"
```

```
N_ITER = 200
N_TRAIN = 5000
LEARNING_RATE = 0.1
PRINT_ITERATIONS = 1

loss_df = {}
train_acc = {}
test_acc = {}
for m in np.array((25, 5, 2, 1)):
    for r in np.array((10, 1, 0.1, 0.01)):
        model = functions_CCNN.ccnn(input_file=INPUT_FILE,
                            output_file=OUTPUT_FILE,
                            label_file=LABEL_FILE,
                            n_iter=N_ITER,
                            n_train=N_TRAIN,
                            learning_rate=LEARNING_RATE,
                            nystrom_dim=m,
                            R=r,
                            print_iterations=PRINT_ITERATIONS)
        model.construct_Q()
        model.train()
        params = "m"+str(m)+"r"+str(r)
        loss_df[params] = model.train_history[:,0]
        train_acc[params] = model.train_history[:,1]
        test_acc[params] = model.train_history[:,2]

# Print validation accuracies for each combination
for key in test_acc:
    print(key, np.max(test_acc[key]))

# Save results for the radial data:
import pickle as pkl
pkl.dump(loss_df, open("results/radial_ccnn_loss.pkl", "wb"))
pkl.dump(train_acc, open("results/radial_ccnn_trainacc.pkl", "wb"))
pkl.dump(test_acc, open("results/radial_ccnn_testacc.pkl", "wb"))
```

A.6.5 NN on simulated data

```
""" This module contains functions to:
    1) Simulate Sum of sigmoids data and store scatterplots
    2) Fit fully-connected neural network to sum of sigmoid and store
        ↪ loss,
       accuracy
    3) Simulate Radial data and store scatterplot
    4) Fit fully-connected neural network to radial and store loss,
        ↪ accuracy
"""

# project-specific modules
import functions_simulate_data
import functions_NN_simulations
import matplotlib.pyplot as plt
```

```
import numpy as np
import pickle as pkl

N = 10000
TRAIN_SPLIT = 0.5
SIGNAL_NOISE_RATIO = 2 # Used to determine variance of noise epsilon
P = 10 # dimension for radial basis function
N_SUBSET = 500 # number of data points to plot in scatterplots

########################################
# 1) SUMS OF SIGMOID simulate data
########################################
""" SUM OF SIGMOIDS MODEL"""
# Simulate data
(X_train, X_test,
 Y_train, Y_test,
 Y_train_bin, Y_test_bin) = \
 functions_simulate_data.create_sigmoidsum_data(N=N,
                                                train_split=TRAIN_SPLIT,
                                                signal_noise_ratio=
                                                    ↪ SIGNAL_NOISE_RATIO,
                                                seed = 1234)

########################################
# 2) SUMS OF SIGMOID run neural nets and plot results
########################################
""" SUM OF SIGMOIDS MODEL: BINARY, FITTING NEURAL NETWORK """
EPOCHS = 30
for NR_HIDDEN in ((5, 25, 100)):
    # shallow fully-connected neural network classifier on binary sum of
        ↪ sigmoids data
    baseline = functions_NN_simulations.NeuralNet_binary()
    baseline.setup_model(input_dim = X_train.shape[1], nr_hidden=
        ↪ NR_HIDDEN)
    baseline.fit_model(X_train=X_train, X_test=X_test,
                       Y_train=Y_train_bin, Y_test=Y_test_bin,
                       epochs=EPOCHS)
########################################
# 3) radial model simulate data
########################################
""" RADIAL FUNCTION MODEL"""
# simulate data
(X_train, X_test,
 Y_train, Y_test,
 Y_train_bin, Y_test_bin) = functions_simulate_data.create_radial_data(N=
     ↪ N,
                        p=P,
                        train_split=TRAIN_SPLIT,
                        signal_noise_ratio=SIGNAL_NOISE_RATIO,
                        seed=1234)
########################################
# 4) radial data run neural nets and store results
########################################
EPOCHS = 60
```

```
""" RADIAL MODEL: BINARY, FITTING NEURAL NETWORK """
for NR_HIDDEN in ((5, 25, 100)):
    # Binary classification version
    baseline = functions_NN_simulations.NeuralNet_binary()
    baseline.setup_model(input_dim = X_train.shape[1], nr_hidden=
        ↪ NR_HIDDEN)
    baseline.fit_model(X_train=X_train, X_test=X_test,
                       Y_train=Y_train_bin, Y_test=Y_test_bin,
                       epochs=EPOCHS)
```

References

[1] Y. Zhang, P. Liang, and M. J. Wainwright, "Convexified convolutional neural networks," *arXiv preprint arXiv:1609.01000*, 2016.

[2] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] C. Poultney, S. Chopra, Y. L. Cun *et al.*, "Efficient learning of sparse representations with an energy-based model," in *Advances in neural information processing systems*, 2007, pp. 1137–1144.

[4] Y. Bengio *et al.*, "Learning deep architectures for ai," *Foundations and trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[5] R. Salakhutdinov and G. Hinton, "Deep boltzmann machines," in *Artificial Intelligence and Statistics*, 2009, pp. 448–455.

[6] ML4A, "Neural networks," website source. [Online]. Available: https://ml4a.github.io/ml4a/neural_networks/

[7] P. Veličković, "Deep learning for complete beginners: convolutional neural networks with keras," website source. [Online]. Available: https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

[8] B. Recht, M. Fazel, and P. A. Parrilo, "Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization," *SIAM review*, vol. 52, no. 3, pp. 471–501, 2010.

[9] M. Fazel, "Matrix rank minimization with applications," Ph.D. dissertation, PhD thesis, Stanford University, 2002.

[10] B. Schölkopf, R. Herbrich, and A. J. Smola, "A generalized representer theorem," in *International conference on computational learning theory*. Springer, 2001, pp. 416–426.

[11] C. K. Williams and M. Seeger, "Using the nyström method to speed up kernel machines," in *Advances in neural information processing systems*, 2001, pp. 682–688.

[12] S. Shalev-Shwartz and Y. Singer, "Efficient learning of label ranking by soft projections onto polyhedra," *Journal of Machine Learning Research*, vol. 7, no. Jul, pp. 1567–1599, 2006.

[13] J. Duchi, S. Shalev-Shwartz, Y. Singer, and T. Chandra, "Efficient projections onto the ℓ_1-ball for learning in high dimensions," pp. 272–279, 2008.

[14] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1.

[15] B. Paranjape, "Stop clickbait: Github dataset," supplement to Chakraborty et al. (2016) - Stop Clickbait. [Online]. Available: https://github.com/bhargaviparanjape/clickbait/tree/master/dataset

[16] A. Chakraborty, B. Paranjape, S. Kakarla, and N. Ganguly, "Stop clickbait: Detecting and preventing clickbaits in online news media," in *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*. IEEE, 2016, pp. 9–16.

[17] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[19] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543. [Online]. Available: http://www.aclweb.org/anthology/D14-1162