Spring 4-18-2019

Deep Embedding Kernel

Linh Le

Follow this and additional works at: https://digitalcommons.kennesaw.edu/dataphd_etd

Part of the Computer Sciences Commons, and the Statistics and Probability Commons

Recommended Citation
https://digitalcommons.kennesaw.edu/dataphd_etd/1

This Dissertation is brought to you for free and open access by the Ph.D. in Analytics and Data Science Research Collections at DigitalCommons@Kennesaw State University. It has been accepted for inclusion in Analytics and Data Science Dissertations by an authorized administrator of DigitalCommons@Kennesaw State University. For more information, please contact digitalcommons@kennesaw.edu.
Deep Embedding Kernel

Author: 

Supervisor: 

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Institute of Analytics and Data Science
The Graduate College

April 28, 2019
Thesis/Dissertation Defense Outcome

Name: Tien Linh Le
Email: lle12@students.kennesaw.edu
Program: Ph.D. in Analytics and Data Science

Title: Deep Embedding Kernel

Thesis/Dissertation Defense: [ ] Passed [ ] Failed [ ] Passed With Revisions (attach revisions)

Date: 4/18/2019

Signatures

Ying Xie
Thesis/Dissertation Chair
Digitally signed by Ying Xie
Date: 2019.04.19 14:59:30 -04'00'

Erik E Westlund
Committee Member
Digitally signed by Erik E Westlund
Date: 2019.04.23 15:53:08 -04'00'

Meng Han
Committee Member
Digitally signed by Meng Han
Date: 2019.04.23 15:53:08 -04'00'

Michael McBurnett
Committee Member
Digitally signed by Michael McBurnett
Date: 2019.04.24 08:31:50 -04'00'

[Signature]
Committee Member

[Signature]
Program Director

[Signature]
Department Chair

[Signature]
Graduate Dean

Last Modified 05/26/16
Declaration of Authorship

I, Linh Le, declare that this thesis titled, “Deep Embedding Kernel” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.

- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

- Where I have consulted the published work of others, this is always clearly attributed.

- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

- I have acknowledged all main sources of help.

- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

________________________________________

Date:

________________________________________
“Is that a whine???”

Jennifer Priestley
Kernel methods and deep learning are two major branches of machine learning that have achieved numerous successes in both analytics and artificial intelligence. While having their own unique characteristics, both branches work through mapping data to a feature space that is supposedly more favorable towards the given task. This dissertation addresses the strengths and weaknesses of each mapping method through combining them and forming a family of novel deep architectures that center around the Deep Embedding Kernel (DEK). In short, DEK is a realization of a kernel function through a newly deep architecture. The mapping in DEK is both implicit (like in kernel methods) and learnable (like in deep learning). Prior to DEK, we proposed a less advanced architecture called Deep Kernel for the tasks of classification and visualization. More recently, we integrate DEK with the novel Dual Deep Learning framework to model big unstructured data. Using DEK as a core component, we further propose two machine learning models: Deep Similarity-Enhanced K Nearest Neighbors (DSE-KNN) and Recurrent Embedding Kernel (REK). Both models have their mappings trained towards optimizing data instances’ neighborhoods in the feature space. REK is specifically designed for time series data. Experimental studies throughout the dissertation show that the proposed models have competitive performance to other commonly used and state-of-the-art machine learning models in their given tasks.
Acknowledgements

First, I want to address my deepest appreciation to Dr. Ying Xie, my dissertation chair. Dr. Xie has been an amazing advisor and mentor to me throughout these four years, and without him, this dissertation cannot be finished. I also want to thank the other committee members, Dr. Jennifer Priestley, Dr. Erik Westlund, Dr. Menghan, and Dr. Michael McBurnet, for their times and efforts in helping my finishing this work. Finally, I want to thank all my professors, colleagues, and everyone, for everything that I have learned, every help that I have received, and every enjoyment that I have had during my study.
## Contents

Declaration of Authorship  
Abstract  
Acknowledgements  

1 Introduction  

2 Literature Study  
   2.1 Kernel Method  
   2.2 Deep Learning  
   2.3 Deep Kernel Methods  
   2.4 Deep Metric Learning  
   2.5 Contribution  

3 The Original Deep Kernel  
   3.1 Input Design  
   3.2 Output Design  
   3.3 The Deep Kernel Architecture  
   3.4 Experimental Study  
   3.5 Discussion
3.6 Conclusion ......................................................... 44

4 Unsupervised Deep Kernel for High Dimensional Data Visualization 45
4.1 Introduction ......................................................... 45
4.2 Unsupervised Deep Kernel ......................................... 47
4.3 $V_{3D}$ Measurement .................................................. 49
4.4 Visualization of High Dimensional Data using Unsupervised Deep Kernel .................................................. 51
4.5 Experimental Study .................................................. 52
4.6 Conclusion ......................................................... 56

5 Deep Embedding Kernel 57
5.1 Introduction ......................................................... 57
5.2 Methodology ......................................................... 59
  5.2.1 Kernel Network .................................................... 60
  5.2.2 Embedding Network ............................................. 64
  5.2.3 Overall Design ..................................................... 64
5.3 Training the Deep Embedding Kernel ........................................... 66
5.4 Experiment Study .................................................. 68
  5.4.1 General Classification ............................................ 68
  5.4.2 Data Visualization ............................................... 72
  5.4.3 The Role of the Embedding Network .......................... 72
5.5 Conclusion ......................................................... 75

6 The Deep Embedding Kernel - Dual Deep Learning Framework 77
6.1 Introduction ......................................................... 77
6.2 The Deep Embedding Kernel - Dual Deep Learning Framework 80
6.3 Experimental Study in Facial Recognition 84

7 Deep Similarity-Enhanced K Nearest Neighbors 87
7.1 Introduction 87
7.2 Methodology 90
  7.2.1 Similarity-Enhanced KNN 92
  7.2.2 KNN Loss 93
  7.2.3 DSE-KNN Architecture 95
7.3 Experimental Study: Classification on Medical Data 98
7.4 Conclusion 101

8 Recurrent Embedding Kernel for Time Series Classification 103
8.1 Introduction 103
8.2 Recurrent Neural Networks for Stock Prediction 106
  8.2.1 Vanilla RNN 107
  8.2.2 LSTM 107
  8.2.3 GRU 110
  8.2.4 Empirical Results 110
8.3 Recurrent Embedding Kernel 112
  8.3.1 Similarity-Enhanced KNN and KNN Loss 112
  8.3.2 Kernel Network and Recurrent Embedding Network 114
8.4 Experiments 118
8.5 Conclusion 121

9 Conclusion 123
## List of Figures

2.1 Using Kernel SVM on Non-linearly-separable Data .......................... 11
2.2 A Common Illustration of a Deep Neural Network ......................... 14
2.3 Illustration of the Computation of a Feature Map Cell in a Convolutional Neural Network ......................................................... 16
2.4 An Example of a Complete Convolutional Neural Network. Image retrieved from [http://deeplearning.net/tutorial/lenet.html](http://deeplearning.net/tutorial/lenet.html) .................................................. 17
2.5 The Computational Flow of RNN ................................................... 17
2.6 The Architecture of a LSTM Block. Figure adapted from (Hochreiter and Schmidhuber, 1997) ......................................................... 19
2.7 The Architecture of a GRU Block. Figure adapted from (Chung et al., 2014) ............................................................. 20
2.8 Decision Hyperplanes of Logistic Regression and SVMs ............... 22
2.9 Deep Architectures Proposed in this Dissertation and Their Relationships ................................................................. 32
3.1 An Illustration of the Deep Kernel Architecture .............................. 37
3.2 Visualization with Kernel PCA using Deep Kernel and RBF Kernel 42
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>The WCSS Curve of the Shuttle Data</td>
<td>48</td>
</tr>
<tr>
<td>4.2</td>
<td>Visualization with Kernel PCA using Deep Kernel and RBF Kernel</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>The Trainable Parameters of Deep Kernel Architecture</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>The Trainable Parameters of Deep Embedding Kernel Architecture</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>The Structure of DEK</td>
<td>61</td>
</tr>
<tr>
<td>5.4</td>
<td>The Structure of the Kernel Network</td>
<td>62</td>
</tr>
<tr>
<td>5.5</td>
<td>Illustration of the Local Pairing Strategy</td>
<td>68</td>
</tr>
<tr>
<td>5.6</td>
<td>Visualization with Kernel PCA using Deep Embedding Kernel and RBF Kernel</td>
<td>73</td>
</tr>
<tr>
<td>5.7</td>
<td>Contributions of Embedding Layers to DEK Performance</td>
<td>75</td>
</tr>
<tr>
<td>6.1</td>
<td>The DEK Architecture for Unstructured Data</td>
<td>78</td>
</tr>
<tr>
<td>6.2</td>
<td>Precision-Recall Curve of Google Facenet in the IMFDB Data</td>
<td>79</td>
</tr>
<tr>
<td>6.3</td>
<td>The Deep Embedding Kernel - Dual Deep Learning Framework</td>
<td>81</td>
</tr>
<tr>
<td>6.4</td>
<td>The Deep Embedding Kernel - Dual Deep Learning Framework for Facial Recognition</td>
<td>83</td>
</tr>
<tr>
<td>6.5</td>
<td>Performance of Transfer Learning Models with DEK and DNN/TL on Facial Recognition</td>
<td>86</td>
</tr>
<tr>
<td>7.1</td>
<td>The Decision Making Process of DSE-KNN</td>
<td>91</td>
</tr>
<tr>
<td>7.2</td>
<td>The Decision Making Process of SE-KNN</td>
<td>93</td>
</tr>
<tr>
<td>7.3</td>
<td>Illustration of the Training Process using KNN Loss $t$</td>
<td>97</td>
</tr>
<tr>
<td>8.1</td>
<td>The Computational Flow of RNN for Predicting Daily Stock Price Direction</td>
<td>108</td>
</tr>
</tbody>
</table>
8.2 The Computational Flow of LSTM for Predicting Daily Stock Price Direction .................................................. 109
8.3 The Decision Making Process of Recurrent Embedding Kernel ......................................................... 113
8.4 The Recurrent Embedding Kernel Computational Flow ............................................................... 116
8.5 Illustration of the Training Process of the Recurrent Embedding Kernel ................................................. 117
8.6 Long-Term Trends from 2008 to 2017 of the Six ETFs in Experimental Study ................................................. 119
List of Tables

3.1 Accuracy Rates of Kernels with SVM in Classification . . . . . . . . . . 41
4.1 Comparison of $V_{3D}$ of Dimension Reduction Algorithms . . . . . . 54
5.1 Test Accuracy Rates of Models in Experiments for Classification . . . 71
7.1 Accuracy Rates of Models in Classification on Medical Data . . . 100
8.1 Accuracy Rates of Models in Predicting Stock Daily Price Direction 121
Dedicated to my parents, who have always been beside me, even when we were half the earth apart.
Chapter 1

Introduction

Kernel methods (Hofmann, Schölkopf, and Smola, 2008) and deep learning (Schmidhuber, 2015) are two major branches of machine learning with numerous successes in data analytics and artificial intelligence. Kernel methods have been widely used in pattern recognition (Schölkopf et al., 2004) (Byun and Lee, 2002); on the other hand, many deep learning models, such as AlexNet (Iandola et al., 2016), Google Facenet (Schroff, Kalenichenko, and Philbin, 2015), and ResNet (He et al., 2016), achieved breakthrough performances when they were proposed. In this dissertation, we first carefully studied both the strengths and potential weakness of these two branches of methods. Then, based on the study, we propose a new family of methods that possesses some significant strengths of both methods, meanwhile remedies their individual weaknesses.

Kernel methods center around the kernel trick (Hofmann, Schölkopf, and Smola, 2008) – using a predefined kernel function to implicitly map data to a new feature space. Ideally, in this new feature space, the data is better distributed towards the given task, e.g., in classification, instances of different classes
are linearly separable. The implicit mapping gives kernel methods an advantage of not needing information on the feature space (e.g. dimensionality, or the mapping function) while still able to accomplish the given task. However, this implicit mapping is relatively heuristic in that there is no guarantee that the predefined kernel can actually lead to a better feature space. Hyper-parameter tuning algorithms like grid-search may improve the model performance (i.e. less prediction errors), but this brute-force strategy does not fundamentally solve the problem of using predefined kernels.

Deep learning, on the other hand, utilizes a high number of parameters structured by layers of neural networks to map the data to an explicit feature space with specified dimensionality (Schmidhuber, 2015). The parameters of the network that determine the mapping are typically tuned based on an explicit learning objective. In other words, by deep learning, the mapping of data into high-level representations is directly guided by the given learning objective through some top-down learning processes such as gradient descent. Therefore, learning objectives play critical roles in the quality of mapping. Frequently used learning objectives try to minimize training errors, which may not have the desired generalizability, according to statistical learning theory (Vapnik, 1999). The work in (Tang, 2013) tries to improve generalizability of classification deep models with SoftMax output by using linear Support Vector Machine (SVM) as the output layer, but the computational complexity of integrating SVM into deep learning is high. Another restriction of deep learning is that the dimensionality of the mapped feature space is pre-specified, instead of being learned.

In this dissertation, we try to address the problems of both kernel machines
and deep learning by proposing a new family of analytical methods that is able to utilize the strengths of each method to address the weakness of the other in a unified framework. This family of algorithms center around the Deep Embedding Kernel (DEK). First of all, DEK does not explicitly map data to a feature space with pre-specified dimensionality, nor implicitly map data through a predefined kernel; instead, DEK uses a newly designed deep architecture to represent a learnable kernel. In other words, DEK utilizes the learning power of deep learning to train a kernel, which in turn implicitly maps data to a high dimensional feature space. The learning objective of DEK specifies a desired relationship of data in the mapped feature space. Then, the kernel represented by DEK, trained by the learning objective, is expected to implicitly map data to such a feature space. Therefore, the whole mapped feature space, including its dimensionality, is learned via deep learning. Using a deep architecture to learn a kernel, instead of directly learning the feature space also has the advantages of flexibility in that the learned kernel can be applied to a wide range of supervised learning tasks including identity detection, general classification, and other kernel based machine learning applications.

The architecture of DEK integrates two learning networks, namely kernel networks and embedding networks. The kernel network directly represents the parameterized kernel trained from data, while the embedding network tries to learn optimized data representations to feed into the kernel network. The training of both networks is done in a single gradient descent process with the same learning objective that specifies an optimized relationship of data in the desired feature space. The paper that proposes DEK has recently been accepted
Chapter 1. Introduction

to be published in the journal Neurocomputing, and will be available on April 28th, 2019 (Le and Xie, 2019).

Prior to DEK, we previously proposed a basic version of the kernel network as the Deep Kernel (DK) for classification (Le et al., 2016) and dimension reduction for visualization of data (Xie, Le, and Hao, 2017). While having achieved better classification accuracy than the traditional Radial Basis Function (RBF) kernel (Hofmann, Schölkopf, and Smola, 2008), the DK also has certain disadvantages. First, the deep architecture utilized in the DK, namely the Deep Belief Network (DBN) (Wang and Raj, 2015), is relatively outdated currently. More specifically, the problem of gradient vanishing, one main purpose of using DBN, has been addressed with simpler but more effective techniques like using the Rectified Linear Unit (ReLU) activation function. Moreover, DBN architectures are generally difficult to trained and thus being rarely used in present. Second, the data undergo a non-linear transformation into paired data before being input into the DK. This leads to potential losses of information, and limits the learning capabilities of the DK.

DEK can be extended to work on different types of data by laying itself on top of deep architectures designed for such data. For example, one can use DEK on top of a Convolutional Neural Network (CNN) (LeCun and Bengio, 1995) for image data, Recurrent Neural Network (RNN) (Funahashi and Nakamura, 1993) for sequential data, or the combination of CNN and RNN for video data. By this extension, the particular deep architecture learns vector embedding from the data in the same learning process where embedding network and kernel network of DEK are trained via gradient descent. Moreover, DEK can be used to
boost the learning power of Transfer Learning (Pan and Yang, 2010) by being laid over a trained deep network that outputs vector embedding. However, problems may emerge when users work with big unstructured datasets without the necessary computational resource typically needed in such cases. Moreover, there exist numerous public knowledge (e.g. pretrained deep architectures, public datasets) for tasks like facial recognition or image recognition that can be utilized for the users’ own purpose. For this reason, we propose to aggregate DEK with the Dual Deep Learning (DDL) framework (Xie and Le, 2018) for such cases to form the DEK/DDL architecture. In short, the DDL consists of two deep architectures with two learning phases: data representation learning and data relationship learning, or equivalently, embedding learning and kernel learning. Phase I aims to learn a representation of data that best approximates the true distribution of the data, therefore it utilizes a data-driven deep network. DEK is integrated into phase II of the DDL where it tries to learn the data relationship with respect to the user given task. In other words, phase I deep network learns an embedding space that best represents the data type, on which the DEK learns the data relationship that best fits the task given. If a pretrained network is used in phase I, the DEK/DDL conceptually become a transfer learning model where the DEK adapts a trained model to solve a different task. With the flexibility of being applicable on any data type while being consistent in term of architecture, we believe this is a unified framework to model big data in a variety of applications and contexts.

Essentially, the mapping in DEK and DK is optimized through a learning objective tied to the pairwise relationship of instances in the data. Recently,
we have proposed works in which the mapping is driven by the neighborhood of instances in the feature space instead. In (Le and Xie, 2018a), we use DEK as a core component to develop a novel model that is called Deep Similarity-Enhanced K Nearest Neighbors (DSE-KNN). In brief, we add a decision making layer (similarity-enhanced K Nearest Neighbors - SE-KNN) and propose a novel loss function (KNN Loss) to train DEK so that the neighborhood of instances in the feature space is optimized for SE-KNN to make decision. Additionally, we introduce a version of DSE-KNN called Recurrent Embedding Kernel (REK) that is specifically designed for time series data in (Le and Xie, 2018b). More specifically, we use a Recurrent Neural Network (RNN) in place of the embedding network, and proceed to train the whole architecture using KNN Loss. REK also uses a SE-KNN as the decision making layer.

Throughout this dissertation, we present experimental studies that show DEK and its variant outperform, or have competitive performance, comparing to other state-of-the-art and commonly used machine learning methods (i.e. machine learning models that are widely used and having top performances) in the given tasks.

The contributions of this dissertation are

1. To kernel methods, this research introduces a learnable kernel function that eliminates the needs of optimizing kernels’ hyper-parameters while still ensuring better feature spaces (in terms of lower errors).

2. To deep learning, the research provides an alternative and more generalizable design that uses a kernel machine instead of a generalized linear
model in the output layer, and through that guides the training process of the network.

3. To big data, this research proposes a unified model in which DEK is integrated into the DDL framework. The framework has the capabilities of utilizing public knowledge into solving the given task.

4. To supervised machine learning, specifically for unstructured data, we introduce two analytical models, namely the DSE-KNN and its REK variant. Like DEK, DSE-KNN also enjoys the combined strength of kernel methods and deep learning while mitigating their weaknesses. DSE-KNN algorithm can be applied on different data types, including unstructured data, and different analytical tasks. REK is introduced as a variant of DSE-KNN for time series data, and has showed its superiority over other deep architectures for sequential data.

5. Overall, this research derives a novel framework applicable to common supervised tasks on different data types. With the combined advantages of kernel methods and deep learning while having each individual branches’ drawbacks mitigated, the framework has the potential to become the preferred approach to machine learning.

The rest of this dissertation is organized as follows. In Chapter 2, we review the related literature from which we discuss how our works are different. Chapter 3 and 4 are dedicated to the original DK in the tasks of classification and visualization, respectively. Chapter 5 presents the methodology of DEK.
Chapter 6, 7, and 8, discuss the DEK/DDL framework, DSE-KNN, and REK, respectively. We finally conclude this dissertation in Chapter 9.
Chapter 2

Literature Study

In this section, we briefly review the literature related to this dissertation, and discuss their weaknesses that motivates our developments of the Deep Embedding Kernel family.

2.1 Kernel Method

Kernel methods belong to a family of algorithms in machine learning that uses the kernel trick to model non-linearity in the data (Hofmann, Schölkopf, and Smola, 2008). In brief, using the kernel trick means to implicitly map the data to a new feature space through a kernel function. As described in (Hofmann, Schölkopf, and Smola, 2008), given the set of data instances $X$ and their label $y$, we use a function $K(\cdot)$ to represent the similarity of instances in $X$:

$$K: X \times X \rightarrow \mathbb{R}, \quad (X^{(i)}, X^{(j)}) \rightarrow K(X^{(i)}, X^{(j)}) \quad (2.1)$$

that satisfies $K(X^{(i)}, X^{(j)}) = \Phi(X^{(i)}) \cdot \Phi(X^{(j)}) \quad \forall \quad X^{(i)}, X^{(j)} \in X$, with $\Phi(\cdot)$ being a mapping of $X$ into a dot product space or feature space, and $X^{(i)}$ and
$X^{(i)}$ being two instances in $X$. We usually consider $K(\cdot)$ as a kernel function that is required to be symmetric and positive-definite. The use of a kernel function allows data algorithms to run in the feature space without explicitly knowing the mapping $\Phi(\cdot)$. Two among the popular kernels are the polynomial kernel

$$K(X^{(i)},X^{(j)}) = (X^{(i)} \cdot X^{(j)} + c)^d$$ (2.2)

and the Radial Basis Function (RBF) or Gaussian kernel

$$K(X^{(i)},X^{(j)}) = \exp(-\gamma ||X^{(i)} - X^{(j)}||)$$ (2.3)

where $c$, $d$, and $\gamma$, are hyperparameters of the kernels.

Perhaps, the most famous supervised kernel method is Support Vector Machine (SVM) (Hofmann, Schölkopf, and Smola, 2008). In the simplest case of binary classification, a SVM seeks a hyperplane that separates the instances in each class while having the maximized margin between the two classes. Let $y^{(i)} \in \{-1, 1\}$ be the label of instance $X^{(i)}$, then the hyperplane $H : W \cdot x + b = 0$ is the solution to the problem

$$\min \ W \cdot W^T + C \sum_{i=1}^{n} \xi^{(i)}$$

s.t. $y^{(i)} (W \cdot X^{(i)} + b) \geq 1 - \xi^{(i)} \ \ \forall \ i \in \{1 \ldots n\}$ (2.4)

where $C$ is a regularization term, and $\xi^{(i)}$'s are slack variables. Solving the problem (2.1) requires the uses of pairwise similarities $s(X^{(i)},X^{(j)})$ among the data. In linear SVM, $s(X^{(i)},X^{(j)}) = X^{(i)} \cdot X^{(j)}$. In data where instances are not linearly separable in their original space, kernel SVM can be utilized. Kernel SVM
uses $K(X^{(i)}, X^{(j)})$ to represent $s(X^{(i)}, X^{(j)})$. Figure 2.1 illustrates the use of using kernel SVM on data where classes are not linearly separable.

![Figure 2.1: Using Kernel SVM on Non-linearly-separable Data](image)

In unsupervised kernel methods, a commonly used model is kernel Principal Component Analysis (kPCA) (Hofmann, Schölkopf, and Smola, 2008). Originally, a linear PCA seeks a transformation of the data into an orthogonal space, usually with fewer dimensions (Pearson, 1901). Given the data $X$, PCA is done by first computing the covariance matrix $\Sigma$

$$\Sigma = \frac{1}{n} \sum_{i=1}^{n} (X^{(i)} \cdot X^{(i)T})$$  \hfill (2.5)

The mapping of $X$ to a new space of $k$ dimensions is retrieved by projecting $X$ on the $k$ eigenvectors associated the highest eigenvalues of $C$. Kernel PCA replaces the covariance matrix $\Sigma$ by

$$\hat{\Sigma} = \frac{1}{n} K(X, X)$$  \hfill (2.6)

with $K(X, X)$ being the kernel matrix computed by using the kernel function
\(K(\cdot): K(X, X)_{i,j} = K(X^{(i)}, X^{(j)})\). PCA and kernel PCA are widely used in dimensional reduction where high-dimensional data are mapped to a 2-dimensional or 3-dimensional space for visualization.

As seen from equations (2.2) and (2.3), commonly used kernels (namely polynomial kernel and RBF kernel) have their forms predefined with some adjustable hyper-parameters \((c, d, \text{ or } \gamma)\). The uses of such kernel functions lead to some inherent weaknesses of kernel methods.

First, the predefined form of the kernels is generally not related to the data. In other words, the form of the kernels is unchanged regardless of situation, and may not be able to reflect the true pairwise relationship among instances.

Second, the tuning of hyper-parameters is generally done in a brute-force manner. In the example of the Grid-Search algorithm, one can start by specifying a space of hyper-parameters, then fitting multiple kernel machines with all possible combinations in such space to select the configuration yielding the highest prediction accuracy. While this method can boost the model performance (i.e. prediction accuracy), it cannot fundamentally solve the problem of using predefined kernels. Moreover, the best set of hyper-parameters may lie outside of the searched space. Additionally, kernel machines like SVM typically have high data complexity (\(O(n^3)\)) thus resulting in difficulties of repeatedly fitting them on big datasets.

The two discussed weaknesses of kernel methods, being predefined, and inefficient tuning process, lead to a third one: predefined kernels have no mechanism to guarantee that their mappings result in a better feature space. The
whole process of selecting a kernel function and selecting a best set of hyper-
parameters is not guided by a goal that is tied to a specific dataset.

In certain cases, the users can also define their own customized kernel func-
tion that is constructed from their domain knowledge to better fit the data. This
is, however, still a heuristic approach. Moreover, there may be latent features in
the data unknown to the domain experts and thus not being utilized. Therefore,
the optimality of such kernels questionable.

2.2 Deep Learning

Unlike kernel machines, deep learning algorithms use a vast number of parame-
ters stacked by layers to model the data (Schmidhuber, 2015). The simplest form
of a deep network is a deep feed-forward network (or deep neural network -
DNN) (Schmidhuber, 2015). Let $H_i$, $W_i$, and $b_i$ denote the output, the weight
matrix, and the bias vector of hidden layer $i$ respectively, then

$$H_{i+1} = \sigma(W_i \cdot H_i + b_i)$$

(2.7)

with $\sigma(\cdot)$ being an activation function, often in the form of sigmoid, hyperbolic
tangent, or rectified linear function (ReLU). The output layer of a DNN uses
a task-driven output function, e.g., SoftMax for classification tasks, or a linear
function for regression tasks. Figure 2.2 shows a common illustration of a DNN.
The nodes refer to the outputs of the layers, the last nodes in each row repre-
sent the bias of the layers, and the connections between nodes represent to the
weights of the layers. DNNs are usually trained to minimize a predefined cost
function \( L \) using gradient descent. With a loss function \( L \) defined, the network is iteratively updated by

\[
W_i \leftarrow W_i - \alpha \frac{\partial L}{\partial W_i} \\
b_i \leftarrow b_i - \alpha \frac{\partial L}{\partial b_i}
\]

(2.8)

where \( \alpha \) is the learning rate. Commonly, \( L \) varies by the task given to the network. For example, a binary classification DNN uses the binary cross-entropy loss function, a multi-label classification DNN uses the negative log likelihood loss function, whereas a regression DNN uses the mean squared error loss function.

Different deep architectures are designed to accommodate different types of data. For instance, a convolutional neural network (CNN) (LeCun and Bengio, 1995) can be used for image data. The CNN architecture uses a set of filters that
traverse through each input image to generate feature maps, which allows features to be detected regardless of their locations in the image. More specifically, let $H_{kl|ij}$ denote the $(i,j)$ cell of feature map $l$ in layer $k$, $H_{kl|ij}$ refer to the region of cell $(i,j)$ of the feature map $l$ of layer $k$, and $W_{kl}$ and $b_{kl}$ be the weight matrix and bias of filter $l$ of layer $k$, and $F_k$ be the set of feature maps of layer $k$, then

$$ H_{kl|ij} = \sigma\left( \sum_{p \in F_{k-1}} (W_{\{k-1\}l} * H_{\{k-1\}p|[ij]} + b_{\{k-1\}l}) \right) $$  \hspace{1cm} (2.9)$$

It should be noted that the number of feature maps of a layer is equal to the number of filters of its previous layer. Figure 2.3 illustrates the computation for cell $(i,j)$ of feature map $l$ of layer $k$. This process is usually referred to as a convolutional layer.

The feature maps output by a convolutional layer are usually further sub-sampled to reduce their dimensionality and signify the major features in the maps. One common sub-sampling method used in CNN is Max-Pooling:

$$ H'_{kl|ij} = \max(H_{kl|ij}) $$  \hspace{1cm} (2.10)$$

where $H'_{kl|ij}$ is the $(i,j)$ cell of the sub-sampled feature map $l$ of layer $k$.

The convolutional/sub-sampling layer pair can be repeated as needed. Their final output are typically connected to regular neural network layers then the output layer. Figure 2.4 illustrates a simple CNN of two convolutional/sub-sampling layers and one dense neural network layer. Recent successful architectures of CNN include AlexNet (Iandola et al., 2016), VGG Net (Simonyan and Zisserman, 2014), ResNet (He et al., 2016), Google FaceNet (Schroff, Kalenichenko,
and Philbin, 2015), and so on.

Recurrent Neural Networks (RNN) are specifically designed to handle temporal information in sequential data. Currently, the state-of-the-art RNN architectures are vanilla Recurrent Neural Network (RNN) (Funahashi and Nakamura, 1993), Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997), and Gated Recurrent Unit (GRU) (Chung et al., 2014). In vanilla RNN’s, the memory state of the current time point is computed from both the current input and its previous memory state. More formally, given a sequence $X = \{X_0, X_1, \ldots, X_n\}$, the hidden state $U_t$ of $X_t$ (i.e. the state of $X$ at time $t$)
2.2. Deep Learning

FIGURE 2.4: An Example of a Complete Convolutional Neural Network. Image retrieved from [http://deeplearning.net/tutorial/lenet.html](http://deeplearning.net/tutorial/lenet.html)

outputted by the network can be expressed as

\[ U_t = \sigma (W \cdot X_t + R \cdot U_{t-1} + b) \]  \hspace{1cm} (2.11)

where \( W \) and \( R \) are weight matrices of the network; \( b \) is the bias vector of the network; and \( \sigma(\cdot) \) is a selected activation function. RNN can be deepened by either stacking multiple RNN layers or increasing the number of layers to compute \( U_t \) from \( X_t \) and \( U_{t-1} \). The computational flow of RNN is shown in Figure 2.5.

FIGURE 2.5: The Computational Flow of RNN

Since its memory state is updated with the current input at every time point,
vanilla RNN is generally unable to keep long-term memory. LSTM is an improved version of RNN with the design goal of learning to capture both long-term and short-term memories. A LSTM block, shown in Figure 2.6, uses gates to control how much its long-term memory would be updated at each time point. The outputted short-term memory is then computed from the current input, the current long-term memory, and the previous short-term memory. More formally, an LSTM block can be described by the following formula:

\[
Z_t = g(W_Z \cdot X_t + R_Z \cdot U_{t-1} + b_z) \tag{2.12}
\]
\[
i_t = \sigma(W_i \cdot X_t + R_i \cdot U_{t-1} + p_i \times C_{t-1} + b_i) \tag{2.13}
\]
\[
f_t = \sigma(W_f \cdot X_t + R_f \cdot U_{t-1} + p_f \times C_{t-1} + b_f) \tag{2.14}
\]
\[
C_t = i_t \times Z_t + f_t \times C_{t-1} \tag{2.15}
\]
\[
o_t = \sigma(W_o \cdot X_t + R_o \cdot U_{t-1} + p_o \times C_{t-1} + b_o) \tag{2.16}
\]
\[
U_t = o_t \times h(C_t) \tag{2.17}
\]

where \(W_\ast\) and \(R_\ast\) are weight matrices; \(b_\ast\) are bias vectors; \(p_\ast\) are peepholes; \(X_t\), \(U_t\), and \(C_t\) are the LSTM’s input, output, and cell state (i.e. long-term memory) at time point \(t\); \(Z_t\) is the proposed update to the cell state; \(i_t\), \(f_t\), and \(o_t\) are the output of the input gate, forget gate, and output gate, respectively; \(g(\cdot)\) is the input activation, \(\sigma(\cdot)\) is the sigmoid function, and \(h(\cdot)\) is the output activation. The overall architecture of a LSTM block is shown in Figure 2.6.

Compared with vanilla RNN, LSTM introduces a mechanism to learn to capture task-relevant long-term memory. At each time point, the captured long-term memory is expressed as a vector. However, the architecture of an LSTM
2.2. Deep Learning

block is relatively complex, which may make training of the a LSTM-based model difficult and time consuming. GRU can be viewed as an alternative to LSTM that can learn to capture task-relevant long-term memories with a simplified architecture. A GRU block contains only two gates, as shown in Figure 2.7. It can be mathematically described using the following formula:

\[ U_t = (1 - z_t) \times U_{t-1} + z_t \times \tilde{U}_t \]  \hspace{1cm} (2.18)

\[ \tilde{U}_t = g(U_t \cdot X_t + R_U \cdot (r_t \times U_{t-1}) + b_U) \]  \hspace{1cm} (2.19)

\[ z_t = \sigma(W_z \cdot X_t + R_z \cdot U_{t-1} + b_z) \]  \hspace{1cm} (2.20)

\[ r_t = \sigma(W_r \cdot X_t + R_r \cdot U_{t-1} + b_r) \]  \hspace{1cm} (2.21)
where all notations are similar to LSTM, except for $z_t$ and $r_t$, which are the outputs of the update gate and reset gate, respectively.

Before the introduction of the ReLU activation function, deep neural networks typically suffered from the gradient vanishing problem. Specifically, in Equation (2.8), the deeper the layer (compared to the output layer), the closer to 0 the gradient terms reach. Having gradients close to 0 prevent a layer from being effectively trained and may make it stay random at the end of training. This issue prevents the construction of deeper networks (more than three hidden layers). Unsupervised layer-wise training algorithms like Stacked Auto-Encoder (SAE) (Vincent et al., 2010) or Deep Belief Network (DBN) (Wang and Raj, 2015) were usually utilized to solve this problem. SAE architectures stack layers of Auto-Encoders, each of which tries to encode the input into a hidden state, then decode the hidden state:

\[
U^{(i)} = \sigma(W \cdot X^{(i)} + b_v) \\
Z^{(i)} = \sigma(U^{(i)} \cdot W^T + b_h)
\]  

(2.22)

where $U^{(i)}$ is the hidden state of $X^{(i)}$, $Z^{(i)}$ is the reconstructed state; $W$, $b_v$, and $b_h$ are the weights and biases of the auto-encoder layer; $W^T$ is the transpose of...
2.2. Deep Learning

The encoding-decoding process is trained by minimizing the reconstruction error using the mean squared error loss function:

$$L_{ae} = \frac{1}{N} \sum_{i \in \text{data}} (X^{(i)} - Z^{(i)})^2$$  \hspace{1cm} (2.23)

DBN, on the other hand, stacks layers of Restricted Boltzmann Machine (RBM) (Wang and Raj, 2015), each of which tries to minimize an energy function between the visible unit (i.e. input) and hidden unit (i.e. output):

$$E(v, h) = -b^T \cdot v - c^T \cdot h - h^T \cdot W \cdot v$$  \hspace{1cm} (2.24)

where $v$ is the visible unit, $h$ is the hidden unit; $W$, $b$, and $c$ are the weights and biases of the RBM. To train SAE and DBN architectures, each layer is first tuned individually using the loss functions in Eq. (2.23) or Eq. (2.24); this process is usually referred to as pre-training. After pre-training, the whole network can be viewed as a normal DNN and continue to be trained as a whole with a task-related loss function (fine-tuning process).

Regardless of the types, traditional supervised deep architectures solve a task by first mapping the data to a higher-level representation on which a generalized linear model is used. More specifically, binary classification networks use logistic regression, multi-label classification networks use SoftMax regression, and regression networks use a simple linear model. Modeling on top of a high-level representation of the data makes deep architectures extremely powerful in solving any problem. However, it is questionable if these output generalized linear models can still be improved. For example, in (Vapnik, 1999), the authors
compare a logistic regression model with a SVM in binary classification, and show that the regression model does not generalize as well as the SVM to data with new patterns. On one hand, logistic regression seeks a hyperplane that divides the two classes and minimizes the error in the training data. On the other hand, SVMs determine a hyperplane that separates the classes and maximizes the margin between them, while the tolerance toward errors can be controlled with a regularization term. The maximized margin provides SVMs with better generalization capabilities than logistic regression. Figure 2.8 illustrates an example of the decision hyperplanes of a logistic regression model and a SVM. The logistic model tries to minimize the training errors, thus it separates the training data almost perfectly but does not adapt well to new data. The SVM tries to maximize the margin between the two classes with some tolerance to error in the training data, therefore it still gives good performance in the new data.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig2-8}
\caption{Decision Hyperplanes of Logistic Regression and SVMs}
\end{figure}
2.3 Deep Kernel Methods

More closely to deep learning, various attempts were made to stack kernels to form deep architectures in (Zhuang, Tsang, and Hoi, 2011), (Strobl and Visweswaran, 2013), (Jose et al., 2013), (Jiu and Sahbi, 2017), and (Sahbi, 2017). The output of this type of deep architecture is typically a highly nonlinear combination of input kernels. The learning process of stacking kernels involves jointly training a SVM classifier and modifying network weights as well as kernel parameters using gradient descent. Some limitations of these works include 1) using pre-defined kernels (such as RBF kernel) as input neurons limits the flexibility and capacity of learning by the deep architecture; 2) using SVM optimization as the learning objective for training the deep architecture is computationally expensive.

Similarly, stacking SVMs to deepen the model architecture was discussed in (Wiering and Schomaker, 2014). The authors of this work use different SVMs to extract latent features in different subsets of dimensions in the data. A global SVM is then used to aggregate all SVMs to form a final decision layer. However, because of the computational expenses of SVMs, it is not practical to form a deep architecture by simply stacking SVMs. Therefore, the extent to which this type of stacking takes advantages of deep learning is rather limited. Instead of stacking SVMs, the work in (Tang, 2013) tried to improve generalization ability of classification deep models with SoftMax output layer by using linear SVM classifier at the top layer to define the learning objective. But this architecture strictly ties with classification tasks and training a SVM at the top layer is still non-trivial as it requires quadratic programming on a batch of data.
In (Wilson et al., 2016), the authors replace the output layer of a traditional neural network by a Gaussian Process (GP) ((Rasmussen, 2004)). The GP kernel’s hyper-parameters and the deep network’s parameters are then jointly learned. This method suffers from the high complexity of solving the top layer GP for each mini-batch.

2.4 Deep Metric Learning

Supervised metric learning has been researched by numerous authors. Methods that are not deep learning based typically compose a kernel function either heuristically (such as the works in (Ben-Hur and Noble, 2005) and (Qiu and Lane, 2009)), or based on solving an optimization problem (such as the works in (Kandola, Shawe-Taylor, and Cristianini, 2002), (Lanckriet et al., 2004), (Ying, Huang, and Campbell, 2009), and (Cortes, Mohri, and Rostamizadeh, 2012)). For instance, a recent work reported in (Sinha and Duchi, 2016) learns random features for a kernel through solving the following optimization problem

$$\max_{Q \in P} \sum y_i y_j K_Q(x_i, x_j)$$

(2.25)

where \((x_i, x_j)\) and \((y_i, y_j)\) are a pair of instances and their respective labels, \(K_Q(\cdot)\) is the kernel function defined on the distribution of random features \(Q\), and \(P\) is the set of possible distribution of random features. Optimization method is also utilized to learn an optimized Mahalanobis distance function of data
2.4. Deep Metric Learning

(Zadeh, Hosseini, and Sra, 2016):

\[
\min_A \sum_{\text{data}} d_A(x_i,x_j) + \sum_{\text{data}} d_{A^{-1}}(x_i,x_j) \tag{2.26}
\]

where \( A \) is a real and symmetric positive definite matrix. A possible disadvantage of these works is the limitation in the forms of the learned kernel functions, as they do not utilize the representation capability of deep learning.

Computing similarity of data using deep networks is mostly done through using Siamese structures for the purpose of classification or similarity ranking. Simply speaking, a Siamese network is a pair of identical neural networks that output embedding vectors for a pair of instances. In (Bromley et al., 1994) and (Hadsell, Chopra, and LeCun, 2006), the Siamese network is learned through optimizing cross-entropy that is computed from either the distance or cosine similarity of each pair of embedding outputs. In (Han et al., 2015) (Chopra, Hadsell, and LeCun, 2005), the pair of output embedding vectors is fully connected to a metric network that outputs the probability that the two input instances are in the same class. Given the full connection between the Siamese network and the metric network, their output probability is not symmetric. In (Zbon- tar and LeCun, 2015) and (Zagoruyko and Komodakis, 2015), the output of the metric network, which is stacked upon two identical CNNs, is maximized if the two input images are of the same class, and minimized otherwise. Their output similarity value of two images are unbounded and non-symmetric.

In (Le et al., 2016) and (Xie, Le, and Hao, 2017), we proposed a deep architecture that is called deep kernel (DK) for both supervised and unsupervised learning. The purpose of the DK is to learn an optimized kernel function for the
given data. The input of DK is some of those constructing elements of both RBF kernel and Polynomial kernel; the output of the DK is guaranteed to be symmetric. DK aims to learn a complex kernel that has more capacity to represent the given data than traditional kernel functions. Experimental results show that DK outperforms RBF kernel with optimized parameters on both supervised and unsupervised learning (Le et al., 2016) (Xie, Le, and Hao, 2017).

Similarity metrics can also be learned using triplets of instances. Google’s FaceNet uses a cost function that is called triplet loss on facial identification (Schroff, Kalenichenko, and Philbin, 2015). Each evaluation of triplet loss involves selecting three instances \(x_i, x_i^+, x_i^-\) that satisfies the following criteria: \(x_i\) is an anchor point, \(x_i^+\) is another data point with the same class as \(x_i\), \(x_i^-\) is a data point with a different class than \(x_i\), and the following inequality holds.

\[
\|x_i - x_i^+\|_2^2 > \|x_i - x_i^-\|_2^2
\] (2.27)

The deep network is then trying to learn a mapping \(f(\cdot)\) such that

\[
\|f(x_i) - f(x_i^+)\|_2^2 < \|f(x_i) - f(x_i^-)\|_2^2 \quad \forall i
\] (2.28)

Therefore, the learning objective of the deep learning can be expressed as minimize the following cost function:

\[
L = \sum_{i=1}^{N} (\|f(x_i) - f(x_i^+)\|_2^2 - \|f(x_i) - f(x_i^-)\|_2^2 + \alpha)
\] (2.29)

with \(\alpha\) being a margin parameter. The Triplet Loss function was extended to other identity detection tasks such as voice recognition (Bredin, 2017). An
issue with triplet loss based cost function, according to (Hermans, Beyer, and Leibe, 2017), is that the training of the network requires large training data that contains a sufficient amount of triplets that satisfies the described criteria.

Last in this section, we would like to mention transfer learning. In the context of deep learning, transfer learning aims to reuse a deep network that is trained for one application to another relevant task (Pan and Yang, 2010) and (Bengio, 2012). A popular way of doing transfer learning is to replace the decision layer(s) of the trained deep network with a new one that is trained for the new task.

2.5 Contribution

Overall, the contributions of this dissertation are as follows.

1. To kernel methods, we provide a kernel function of which mapping is guided by a learning goal that specifies the desired features of the mapped space. Moreover, the form of the kernel function represented by DEK is not predefined but is learned from data. Overall, DEK provides a kernel function that is learnable from the data, which solves the problem of optimizing the kernel function and its hyper-parameters in traditional kernel methods.

2. To deep learning models, we provide a newly designed architecture that 1) does not map the data to an explicit feature space, but rather to a learnable implicit space without its dimensionality predefined, and 2) has an objective of learning a feature space that has the desired data relationship,
instead of minimizing training error like common deep neural networks, enabling greater generalizability.

3. To big data, we provide a unified framework for users to utilize public knowledge into solving their given tasks through the DEK/DDL framework. The framework can also be applied into transfer learning where pretrained architectures are integrated into the users’ given tasks with the help of DEK.

4. The DSE-KNN algorithm is a contribution to supervised learning, especially on unstructured data. DSE-KNN can be applied on different types of data on different applications by changing its embedding network component. The REK version of DSE-KNN is introduced as a case study of applying DSE-KNN on time series data.

5. Overall, to supervised learning, we introduce a family of novel deep algorithms that is able to outperform current commonly used and state-of-the-art machine learning models in their given tasks, as will be shown in experimental studies throughout this dissertation.

The remainder of the dissertation is proposed to make the following contributions.

- Chapter 3 initializes the idea of learning a kernel function using a novel deep architecture that is call Deep Kernel (DK). DK is tested in classification and visualization; and was compared to the Gaussian kernel in such tasks. The contribution of this chapter was published as:
2.5. Contribution


- Chapter 4 extends the use of DK as an unsupervised method to visualization of high-dimensional data. The contribution of this chapter was published as:

  Ying Xie, Linh Le, and Jie Hao (2017). “Unsupervised deep kernel for high dimensional data”. In: Neural Networks (IJCNN), 2017 International Joint Conference on. IEEE, pp. 294–299

- Chapter 5 addresses the problem of DK and develop the Deep Embedding Kernel (DEK) as an upgraded version of DK. DEK is tested in general classification and visualization against other common machine learning models including Support Vector Machine with Gaussian kernel, Gradient Boosting model, Random Forest, and Neural Network. The contribution of this chapter will be published as:


- Chapter 6 proposes to use DEK with the Dual Deep Learning (DDL) framework for big data. The DEK/DDL framework is tested in facial recognition and achieved better results than Google Facenet. The framework was filed as a provisional patent with Equifax Inc.

• Chapter 7 proposes a new deep model is called Deep Similarity-Enhanced K Nearest Neighbors (DSE KNN) that has DEK as its core component. The model is compared to other machine learning models and deep models in classification of disease data. The contribution of this chapter was published as:


• Chapter 8 proposes a version of DSE-KNN that is called Recurrent Embedding Kernel (REK) for time series data analysis. Experiments were done in predicting next day stock price movements; REK was compared with other common deep models for sequential data, including vanilla Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU). The contribution of this chapter was published as:


The relationship among the architectures proposed in each chapter of this dissertation is shown in Figure 2.9. The potential future developments of this dissertation can be divided into two categories: applying the proposed models
2.5. Contribution

into current challenging problems, or improving the design of the kernel architecture.
Chapter 2. Literature Study

Figure 2.9: Deep Architectures Proposed in This Dissertation and Their Relationships

Dissertation Roadmap
Chapter 3

The Original Deep Kernel

The Deep Kernel (DK) (Le et al., 2016) is our first attempt in addressing the problem of mapping data to feature spaces in kernel methods and deep learning. The DK architecture is designed to map data to an implicit feature space optimized for the given task. In other words, the DK must satisfy the constraints of a true kernel function, which are as follows

• Taking input of two data instances simultaneously, and outputting their similarity as a single scalar. Let \( D(\cdot) \) be the function that is represented by the DK, then

\[
D(X^{(i)}, X^{(j)}) = K(X^{(i)}, X^{(j)}) \in \mathbb{R} \tag{3.1}
\]

• Being symmetric

\[
K(X^{(i)}, X^{(j)}) = K(X^{(j)}, X^{(i)}) \tag{3.2}
\]

• The output kernel matrix \( K \) is positive-definite. \( K \) is constructed with DK as

\[
K \succ 0 \quad | \quad K_{ij} = K(X^{(i)}, X^{(j)}) \tag{3.3}
\]
where $K_{ij}$ is cell $(i, j)$ of $K$.

For classification, a desirable kernel function can map data to a feature space where the similarity of instances in different classes is low while the in-class similarity is high. The next sections discuss our designs of input layer, output layer, and network architecture, so that DK can be considered a true kernel function represented by a deep neural network.

### 3.1 Input Design

We apply a transformation $f(\cdot)$ on two data instances $X^{(i)}$ and $X^{(j)}$ so that they become a single input vector for the DK. Equation (3.1) then becomes

$$K(X^{(i)}, X^{(j)}) = D(f(X^{(i)}, X^{(j)}))$$  \hspace{1cm} (3.4)

The transformation $f(\cdot)$ must also be designed to satisfy the symmetricity constraint of a kernel function in Eq. (3.2). We take into consideration the common kernels such as polynomial kernel and RBF kernel when designing $f(\cdot)$. More specifically, expanding the polynomial kernel gives

$$K_{poly}(X^{(i)}, X^{(j)}) = (X^{(i)} \cdot X^{(j)} + c)^d = \sum_{k=1}^{d} f_k(X_k^{(i)} \ast X_k^{(j)}) + C$$  \hspace{1cm} (3.5)

and expanding the RBF kernel gives

$$K_{RBF}(X^{(i)}, X^{(j)}) = \exp(-\gamma \|X^{(i)} - X^{(j)}\|_2^2) = \exp\left(\sum_{k=1}^{d} f_k(|X_k^{(i)} - X_k^{(j)}|)\right)$$  \hspace{1cm} (3.6)
with $d$ being the number of dimensions of the data, $X_k^{(i)}$ being the $k^{th}$ feature of $X^{(i)}$. It can be seen that the two kernels are functions of the element-wise products or the differences for each dimension of the original data. Based on this observation, we design $f(\cdot)$ as follows

$$f(X^{(i)}, X^{(j)}) = \begin{cases} \prod_{k=1}^{d} X_k^{(i)} * X_k^{(j)}, & \text{if } \text{element-wise product} \\ \prod_{k=1}^{d} \left| X_k^{(i)} - X_k^{(j)} \right|, & \text{if } \text{element-wise difference} \end{cases}$$

(3.7)

In other words, similarity data is input into DK at the finest level of granularity so that the network has the most flexibility to optimize the kernel function based on the given task. Moreover, this design of $f(\cdot)$ ensures the symmetricity of the kernel function represented by DK.

### 3.2 Output Design

As mentioned, for classification, the kernel function needs to separate data instances in different classes in the feature space. One way to do this is to set the training goal of DK to maximize the output similarity of instances in the same classes while minimizing that of instances in different classes. Accordingly, we use the \textit{sigmoid} activation in the output layer of DK, and train the networking using the Binary Cross-Entropy loss function.

$$D(f(X^{(i)}, X^{(j)})) = \frac{1}{1 - \exp(W_k \cdot H_k^{(i,j)} + b_k)}$$

(3.8)
Chapter 3. The Original Deep Kernel

\[
L = \sum_{data} (Y^{(i,j)} \log(K(X^{(i)}, X^{(j)}) + (1 - Y^{(i,j)}) \log(1 - K(X^{(i)}, X^{(j)}))) \quad (3.9)
\]

where \(W_k\) and \(b_k\) are the weights and biases of the output layer of DK, \(H^{(i,j)}_k\) is the output of the last hidden layer of DK computed from \(X^{(i)}\) and \(X^{(j)}\), and \(Y^{(i,j)}\) is the target similarity of \(X^{(i)}\) and \(X^{(j)}\). The computation of \(Y^{(i,j)}\) is further explained in section 3.3.

This selection of output function and loss function further provides an interpretation of the similarity between two instances: how similar two instances are is measured by the probability of them belonging to the same classes:

\[
K(X^{(i)}, X^{(j)}) = P(y^{(i)} = y^{(j)} | X^{(i)}, X^{(j)}) \quad (3.10)
\]

In other words, the higher similarity two instances have, the higher probability of them belonging to the same class.

Furthermore, the use of sigmoid output function also constrains the output of DK, \(K(X^{(i)}, X^{(j)}) \in (0, 1)\), through which ensuring the positive-definiteness of the output kernel matrix \(K\).

### 3.3 The Deep Kernel Architecture

The overall structure of the DK is illustrated in Figure 3.1. Two instances \(X^{(i)}\) and \(X^{(j)}\) are transformed into a single vector of dimension-wise similarities (including products and absolute differences). The input vector is fed into the DK which outputs a single scalar \(K(X^{(i)}, X^{(j)})\) as the similarity of \(X^{(i)}\) and \(X^{(j)}\).
3.3. The Deep Kernel Architecture

We further utilize the DBN architecture to improve the learning capability of DK. More specifically, each layer of DK is first pre-trained as an RBM before the whole network is fine-tuned using the loss function in Equation (3.9). All layers of DK use sigmoid activation as this is required by RBM’s.

While there are no rules on setting the sizes of hidden layers, we fix the numbers of neurons in each layer to reduce the number of hyper-parameters of the network. Let the dimensionality of the original data be $d$, the size of the $k^{th}$ hidden layer be $N_k$, and the number of hidden layers be $m$, then

$$N_1 = N_2 = \cdots = N_m = 2 \times d \quad (3.11)$$
To fine-tune DK, we first form the training data which consists of pairs of instances. As mentioned, the paired input $f(X^{(i)}, X^{(j)})$ is formed using Equation (3.7). We further compute the target similarity $Y^{(i,j)}$ of each pair using their original classes ($y^{(i)}$ and $y^{(j)}$) as follows

$$\begin{align*}
Y^{(i,j)} &= 1 \iff y^{(i)} = y^{(j)} \\
Y^{(i,j)} &= 0 \iff y^{(i)} \neq y^{(j)}
\end{align*}$$

(3.12)

More specifically, if two instances belong to the same class, their target similarity is 1, otherwise it is 0. Overall, each pair of instances $X^{(i)}$ and $X^{(j)}$ forms a training instance $(f(X^{(i)}, X^{(j)}), Y^{(i,j)})$ for DK.

The network is trained with the Binary Cross-Entropy loss function as in Equation (3.9). As can be seen, optimizing the network using Equation (3.9) maximizes $K(X^{(i)}, X^{(j)})$ if $Y^{(i,j)} = 1$, and minimizes $K(X^{(i)}, X^{(j)})$ if $Y^{(i,j)} = 0$. In other words, a DK learns to map the data to a feature space where the similarity of instances in the same classes is close to 1, while the similarity of instances in different classes is close to 0. Intuitively, the kernel network is trained to implicitly map data to a feature space where instances in the same classes have high similarity and instances in different classes have low similarity. Similar to other neural networks, the weight matrices and bias vectors of the DK can then be updated via Gradient Descent:

$$\begin{align*}
W^{(i)} &\leftarrow -\alpha \ast \frac{\partial L}{\partial W^{(i)}} \\
b^{(i)} &\leftarrow -\alpha \ast \frac{\partial L}{\partial b^{(i)}}
\end{align*}$$

(3.13)
where \( W^{(i)} \) and \( b^{(i)} \) are the weights and bias of layer \( i \), and \( \alpha \) is the learning rate. We summarize the whole training process of DK in Algorithm 1.

### Algorithm 1 Training Deep Kernel

**Input** dataset \((X, y)\) of \(n\) instances, number of pre-training epochs \(e_p\), number of fine-tuning epochs \(e_f\).

1. Initializing training set: \( X \leftarrow \emptyset; \quad Y \leftarrow \emptyset \)  \(\triangleright\) Forming paired data
2. for each pair of instances \((i, j)\) do \(\triangleright\) Populating paired data
   3. Append \(f(X^{(i)}, X^{(j)})\) to \(X\)
   4. Append \(Y^{(i,j)} = (y^{(i)} = y^{(j)})\) to \(Y\)
5. Randomly initializing weights and biases of each layer of DK
6. for each layer of DK do \(\triangleright\) Pre-training
   7. Train layer as RBM using \(X\) for \(e_p\) iterations
8. Fine-tune DK using \((X, Y)\) for \(e_f\) iterations
9. return trained DK

### 3.4 Experimental Study

We test the DK in two tasks: classification and dimension reduction for visualization.

In the classification task, the trained DK is utilized as a kernel function for SVM (SVM/DK). The SVM/DK models are compared with SVM using the regular RBF kernels (SVM/RBF). Experiments with the two models are conducted with six datasets, including

1. Breast Cancer (Michalski et al., 1986). The data consists of 699 instances with 10 attributes of breast tumors. The goal is to determine if the tumor is benign or malignant.
2. Breast Cancer (Diagnostic) (Wolberg and Mangasarian, 1990). The original data that was used to derive the Breast Cancer data above. There are 32 attributes in this dataset.

3. Wine Quality (Aeberhard, Coomans, and De Vel, 1992). This dataset consists of 12 attributes towards the characteristics of 4898 wine samples. The goal is to determine the quality score (ranked from 0 to 10) of each wine sample. This can be either a regression or a classification problem, we consider this to be a classification task.

4. Segment (Zhang, 1992). The data consists of 2310 instances with 19 attributes. Each instance was originally an outdoor image of which features were manually extracted by the researchers. There are seven classes in the data.

5. Cardiotocography (Campos et al., 2000). The data consists of 23 features extracted from 2126 cardiotocograms. The data was categorized into 10 classes by three expert obstetricians.

6. Pima Indian Diabetes (Smith et al., 1988). The data was collected from the Pima Indian population near Phoenix, Arizona. Eight dimensions of the data include number of times pregnant, plasma glucose concentration from an oral glucose tolerance test, diastolic blood pressure, triceps skin fold thickness, 2-hour serum insulin, body mass index, diabetes pedigree function, and age. The class label is whether the patients had diabetes or not. The dataset has 768 instances.
3.4. Experimental Study

All datasets are split into training sets (70% instances) and test sets (30% instances).

To train the DK’s, paired data is generated from the training sets of the original data, then further divided into a training set (70%), validation set (15%), and testing set (15%). DK’s are trained using early stopping criteria: if the misclassification rate does not improve within a certain number of iterations, then the training stops. Otherwise, the maximum number of training epochs is set to be 1500. All DK’s have eight hidden layers of which each has $2d$ neurons, with $d$ being the dimensionality of the input.

Other hyper-parameters like SVM’s regularization term $C$ (see Equation 2.1), and RBF kernel’s $\gamma$ (see Equation 2.3), are optimized with grid search.

Table 3.1 shows the classification accuracy of SVM/DK and SVM/RBF in six tested datasets. It can be seen that DK outperforms RBF kernels in all experiments.

**Table 3.1: Accuracy Rates of Kernels with SVM in Classification**

<table>
<thead>
<tr>
<th>Data</th>
<th>No. of Classes</th>
<th>SVM/DK</th>
<th>SVM/RBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breast Cancer</td>
<td>2</td>
<td>98.05</td>
<td>97.32</td>
</tr>
<tr>
<td>Breast Cancer (Diagnostic)</td>
<td>2</td>
<td>98.24</td>
<td>97.66</td>
</tr>
<tr>
<td>Wine Quality</td>
<td>11</td>
<td>59.17</td>
<td>57.29</td>
</tr>
<tr>
<td>Segment</td>
<td>7</td>
<td>97.14</td>
<td>95.84</td>
</tr>
<tr>
<td>Cardiotocography</td>
<td>2</td>
<td>99.37</td>
<td>97.18</td>
</tr>
<tr>
<td>Pima Indian Diabetes</td>
<td>6</td>
<td>78.35</td>
<td>75.32</td>
</tr>
</tbody>
</table>

Experiments on dimensional reduction and visualization are done on the Breast Cancer and the Segment data. We map the Breast Cancer into a 2-dimensional space, and the Segment data into both 2-dimensional and 3-dimensional spaces.
Figure 3.2: Visualization with Kernel PCA using Deep Kernel and RBF Kernel
Figure 3.2 shows the visualization in 2-dimensional and 3-dimensional spaces of the Breast Cancer and Segment data using the trained DK and RBF as kernel functions in kernel PCA. Each dot represents a data instance, and the dots’ colors represent their classes. It can be seen that, in the feature spaces mapped by DK, instances in the same classes are closer than in the spaces mapped by the RBF kernel. Furthermore, the clusters of classes in the spaces mapped by DK are more linearly-separable.

3.5 Discussion

In classification, DK outperforms RBF kernel in all experiments. This shows the advantage of having a kernel function trained from the data instead of predefined. However, an issue on the proposed deep kernel method is the time and data complexity of the method. From the time perspective, DK is a deep network and suffers from the longer training time like other deep learning methods. From the data perspective, DK is trained with pairs of instances which means a training of size $n$ results in a data complexity of $O(n^2)$. This yields difficulties if DK is to be applied on big data. Like with other deep learning methods, high performance systems, preferably with Graphic Processing Units (GPU), should be utilized for the training process. We also propose different sampling strategies on paired data in following chapters.

In dimensional reduction for visualization, DKs also outperforms RBF kernels. It can be seen from Figure 3.2 that DK maps data to a feature space where instances in the same class are in a more compacted area. It is also visibly that the
classes’ cluster mapped by DKs are more linearly separable in low dimensional spaces compared to those mapped by RBF kernels. This is, however, a supervised dimensional reduction method. In Chapter 4, we discuss the unsupervised version of DK for visualization of data, and formally propose a measurement for the performance of unsupervised dimension reduction model’s performance.

3.6 Conclusion

This chapter proposes the Deep Kernel architecture, a realization of a kernel function through a DNN. DK is a kernel function that is trainable from the data and thus solving the problem of pre-defined kernel functions in kernel methods. Experimental results show that a DK outperforms a RBF kernel with parameters optimized via Grid Search. DK is also a potential approach in dimensional reduction and visualization given a training set is available. Experiments show that visualizations generated using DK are visibly better than that of RBF kernels.
Chapter 4

Unsupervised Deep Kernel for High Dimensional Data Visualization

4.1 Introduction

Visualization of high-dimensional data is always beneficial in data analytics. A high-quality visualization might help users select proper analytical methods or make decisions. While numerous novel techniques have been proposed for visualizing big and high-dimensional data, many of them require users to have specific skills to interpret the visualization. Reducing high-dimensional data into a 3-dimensional space for visualization is still among the more intuitive methods. In this chapter, we propose an algorithm to train the Deep Kernel (DK) in an unsupervised manner for high-dimensional data without labels. The trained unsupervised DK is then used as a kernel function for kernel PCA to reduce the high-dimensional data into a 3-dimensional space for visualization. The algorithm is also designed so that there are minimal changes in structural information between the original data and the reduced data. More specifically, we
aim to preserve the cluster structure of the data in the reduced space compared to the original space with the unsupervised Deep Kernel.

As mentioned in Chapter 2, PCA and kernel PCA are two common methods used in dimension reduction and visualization. Both map data to an orthogonal space. For visualization, the mapped spaces have two or three dimensions. Another approach, Kernel Entropy Components Analysis (KECA) (Jenssen, 2010) aims to maximize the Renyi quadratic entropy in the input space for dimension reduction. Deep architectures such as the DBN or the Stacked Auto-Encoders (SAE) (Wang and Raj, 2015), have also been used widely in the dimension reduction tasks. These approaches make use of the unsupervised representation ability of the stacked Restricted Boltzmann Machines (RBM) or Auto-Encoders (Wang and Raj, 2015) to map the data to the reduced space. For examples, DBN is used for dimension reduction on the AR Face Database in (Noulas and Krose, 2008); SAE is applied on the MNIST, Olivetti Face Data, and synthesized data in (Wang et al., 2015).

To ensure the reliability of the visualization on a 3-dimensional space, we use the $V_{3D}$ measurement in (Xie et al., 2016) to evaluate the amount of structural information maintained by the dimension reduction process.

To evaluate the unsupervised Deep Kernel, we compare our proposed method to the mentioned dimension reduction approaches on different datasets. Experimental studies show that the unsupervised DK outperforms all other methods in dimension reductions with respect to the $V_{3D}$ measure.
4.2 Unsupervised Deep Kernel

The DK architecture in this chapter is identical to the one discussed in Chapter 3. However, we focus on developing an algorithm to train DK in an unsupervised manner for dimension reduction of unlabeled data rather than in a supervised context.

DK is originally proposed under a supervised setting, therefore the training process requires data labels. As the designed goal of the unsupervised DK is for dimension reduction with minimal changes in the cluster structure of data, we use clustering techniques on the data to obtain the cluster labels which are then used as training labels for DK.

There have been many proposed clustering methods, but most are not scalable to big data Therefore, we propose to use K-Means (Ng, 2012) for this task. Although the K-Means algorithm is generally accepted as being simple, it is effective and scalable to large data sets. Also, platforms such as Hadoop/Spark allow K-Means to be parallelized effectively, making it more adaptive to big data. Using K-Means requires choosing a hyper-parameter $k$ indicating the number of clusters. A common way of choosing $k$ is to use the within-cluster sum of squares (WCSS) criterion (Krzanowski and Lai, 1988). Mathematically, WCSS can be expressed as

$$ WCSS = \sum_{i=1}^{k} \sum_{X^{(j)} \in C_i} ||X^{(j)} - \mu_i||^2 $$  \hspace{1cm} (4.1)

where $\mu_i$ is the mean vector of the cluster $C_i$, and $X^{(j)}$'s are the instances. In general, as $k$ increases, WCSS decreases and become 0 when $k = N$, with $N$
being the data size, therefore an optimal $k$ cannot be chosen to minimize $\text{WCSS}$.
Instead, an optimal number of clusters can be selected by applying K-means on
data with different $k$ values, then choosing the $k$ value where the $\text{WCSS}$ meets
an elbow point (point with noticeably sharp decrease in the change of $\text{WCSS}$).
Figure 4.1 shows an example of an elbow point in $\text{WCSS}$ when increasing $k$ from
2 to 14. In this case, $k = 4$ is chosen to be a good cluster structure for the data.
The example is built from the Shuttle data set (Michie, 1995).

![Figure 4.1: The WCSS Curve of the Shuttle Data](image)

We apply K-Means with the selected $k$ on the data to obtain the cluster labels
and use them as training labels for DK. With this labeling strategy, the unsupervised
DK is trained to map data to a feature space where the similarity of
instances within a cluster is maximized; this similarity is minimized otherwise.
In other words, the unsupervised DK can be used in dimension reduction and
4.3. **V$^{3D}$ Measurement**

We proposed the V$^{3D}$ measurement in (Xie et al., 2016) to evaluate the quality of dimension reduction methods. In brief, V$^{3D}$ evaluates how much the original clustering structure is preserved when data undergoes dimension reduction to a 3-dimensional space.

Given a high dimensional data set $D_{org}$, we assume that $D_{org}$ has an optimal clustering structure with $k$ clusters in its original space, denoted as $C_{org} = \{C_1, C_2, \ldots, C_k\}$. Then, we reduce the dimensionality of $D_{org}$ to three dimensions to obtain the reduced data $D_{3D}$. Using the same clustering method as for $D_{org}$, we further generate $k$ clusters on $D_{3D}$: $C_{3D} = \{C'_1, C'_2, \ldots, C'_k\}$. Next, we calculate $H(C_{3D}|C_{org})$ - the entropy (Gray, 2011) of $C_{3D}$ with respect to $C_{org}$. $H(C_{3D}|C_{org})$ reflects how much the clustering structure changes when reducing $D_{org}$ to $D_{3D}$. More specifically

$$H(C_{3D}|C_{org}) = \frac{\sum_{i=1}^{k} (|C'_i| * H(C'_i|C_{org}))}{\sum_{i=1}^{k} |C'_i|}$$ (4.2)

where

$$H(C'_i|C_{org}) = -\sum_{j=1}^{k} \frac{|\{X^{(*)}|X^{(*)} \in C'_i \cap \{X^{(*)} \in C_j\}|}{|\{X^{(*)}|X^{(*)} \in C'_i\}|} * \log_2 \frac{|\{X^{(*)}|X^{(*)} \in C'_i \cap \{X^{(*)} \in C_j\}|}{|\{X^{(*)}|X^{(*)} \in C'_i\}|}$$ (4.3)
with $X^{(s)}$ being an instance in the data. Finally, $V_{3D}$ can be obtained as follows

$$V_{3D} = 1 - H(C_{3D}|C_{org})$$

(4.4)

As can be seen, $V_{3D}$ measures how similar the original cluster structure and the reduced cluster structure are. In other words, $V_{3D}$ indicates how much the the original cluster structure is preserved when the data is reduced into a 3-dimensional space.

In (Gray, 2011), the author show that $0 \leq H(C_{3D}|C_{org}) \leq \log_2(k)$; the minimum value 0 occur when all instances belong to a cluster in the original space also belong to a same cluster in the reduced space. Therefore, $V_{3D}$ has a maximum value of 1 which occurs when the original clustering structure is perfectly preserved during dimension reduction.

Besides utilizing $V_{3D}$ in comparing dimension reduction methods, we also use $V_{3D}$ in determine if a dataset can be effectively reduced and visualized in a 3-dimensional space. More specifically, we select a threshold $\alpha$ for $V_{3D}$ (e.g. $\alpha = 0.75$) and only visualize the data if $V_{3D} \geq \alpha$ after dimension reduction. A $V_{3D} < \alpha$ means that the level of how much the original clustering structure is preserved during dimension reduction is not acceptable.
4.4 Visualization of High Dimensional Data using Unsupervised Deep Kernel

The entire process of visualizing a given high-dimensional dataset $X$ using unsupervised DK is as follows

1. Select a threshold $\alpha$ as the minimal value of $V_{3D}$ that is considered acceptable for visualization

2. Determine an optimal clustering structure for the original data

   2.1. Apply K-Means on the original data $X$ with different $k$ values selected from a range of interests.

   2.2. Use $W_{CSS}$ of each $K-Means$ model to select the optimal $k = k^*$

   2.3. Apply K-Means with the optimal $k^*$ on $X$ to obtain optimal clustering structure $C_{org}$

3. Train the unsupervised DK

   3.1. Form the training data for DK

       • For each pair $(X^{(i)}, X^{(j)})$, add $X^{(i,j)}$ as presented in Equation 3.7 to the training feature set

       • For each pair $(X^{(i)}, X^{(j)})$, add $y^{(i,j)} = (C^{(i)}_{org} = C^{(j)}_{org})$ to the training label set ($C^{(i)}$ is the cluster label of $X^{(i)}$). In other words, $y^{(i,j)} = 1$ if $(C^{(i)}_{org} = C^{(j)}_{org})$, and $y^{(i,j)} = 0$ otherwise.

   3.2. Train the unsupervised DK using the training feature set and the training label set
4. Use the trained DK as the kernel function for kPCA to reduce the dimensionality of $X$ to three to obtain $X_{3D}$

5. Apply K-Means on $X_{3D}$ with $k = k^*$ to obtain the new clustering structure $C_{3D}$

6. Compute $V_{3D}$ of the process

7. If $(V_{3D} \geq \alpha)$ then visualize the data; otherwise the data cannot be reduced to a 3-dimensional space with acceptable preservation in clustering structure.

### 4.5 Experimental Study

In this section, we conduct experimental studies to evaluate the performance of the unsupervised DK along with kPCA in the dimension reduction of data into a 3-dimensional space for visualization. Five datasets are used in the experiment, including

1. Ionosphere (Sigillito et al., 1989). The data was processed signals collected by a radar system in Goose Bay, Labrador. There are 34 attributes in the data.

2. Wine (Aeberhard, Coomans, and De Vel, 1992) (Aeberhard, Coomans, and Vel, 1992). This dataset consists of 12 attributes towards the characteristics of 4898 wine samples.

3. Shuttle (Michie, 1995). The data consists of nine attributes referring to characteristics of 58,000 aircraft instances.
4. Ecoli (Horton and Nakai, 1996). The data consists of 8 attributes referring to protein localization sites of 336 Ecoli instances.


While the certain datasets have label information, we omit the uses of labels in all experiments since this study focuses on unsupervised dimension reduction for visualization.

We aggregate the performance of unsupervised DK to the results of other methods reported in (Xie et al., 2016) which include PCA, kPCA with RBF kernel (GK-PCA), kPCA with Polynomial Kernel (PK-PCA), kECA using RBF kernel (GK-ECA), kECA using Polynomial kernel (PK-ECA), Singular Value Decomposition (SVD), Deep Belief Network (DBN), and Stacked Auto-Encoder (SAE). We use $V_{3D}$ to evaluate the quality of the dimension reduction algorithms.

Following the process that is described in Section 4.4, we first determine the optimal cluster structure of the given data using the K-Means/WCSS method. We then generate the paired data for DK using the cluster labels, and split the paired data into a training set (50%), validation set (25%), and testing set (25%). In all experiments, the DK’s have eight hidden layer layers. The trained DK’s are then used as kernel functions in kPCA to reduce the dimensionality of the given data to three. Finally, we perform K-Means clustering on the reduced data to compute $V_{3D}$ for unsupervised DK.

Table 4.1 shows the $V_{3D}$ measurements of all methods. As can be seen, kPCA
using unsupervised DK outperforms all other methods including PCA (previously the champion model) in all datasets. Moreover, in all experiments, unsupervised DK results in a $V_{3D}$ of 1, indicating that dimension reduction using DK perfectly preserves the cluster structure of the original data in the reduced space.

Table 4.1: Comparison of $V_{3D}$ of Dimension Reduction Algorithms

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Dim.</th>
<th>Opt. $k$</th>
<th>PCA</th>
<th>DK-PCA</th>
<th>GK-PCA</th>
<th>PK-PCA</th>
<th>GK-ECA</th>
<th>PK-ECA</th>
<th>SVD</th>
<th>DBN</th>
<th>SAE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ionosphere</td>
<td>34</td>
<td>2</td>
<td>0.98</td>
<td>1</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.97</td>
<td>0.78</td>
<td>0.70</td>
<td>0.71</td>
</tr>
<tr>
<td>Wine</td>
<td>13</td>
<td>3</td>
<td>0.93</td>
<td>1</td>
<td>0.81</td>
<td>0.81</td>
<td>0.78</td>
<td>0.81</td>
<td>0.84</td>
<td>0.66</td>
<td>0.69</td>
</tr>
<tr>
<td>Shuttle</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.70</td>
<td>0.75</td>
<td>0.79</td>
</tr>
<tr>
<td>Ecoli</td>
<td>8</td>
<td>3</td>
<td>0.98</td>
<td>1</td>
<td>0.94</td>
<td>0.94</td>
<td>0.44</td>
<td>0.93</td>
<td>0.73</td>
<td>0.77</td>
<td>0.81</td>
</tr>
<tr>
<td>B. Cancer</td>
<td>32</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.94</td>
<td>0.94</td>
<td>0.94</td>
</tr>
</tbody>
</table>

We further visualize the datasets with DK-PCA and PCA, and show the results in Figure 4.2. As can be seen, clusters in the 3-dimensional spaces generated by DK are more compacted, and their inter-cluster distances are larger than those generated by PCA.
Figure 4.2: Visualization with Kernel PCA using Deep Kernel and RBF Kernel
4.6 Conclusion

This chapter presents an unsupervised deep kernel approach to visualize high dimensional data in a 3-dimensional space. The overall process begins with finding an optimal cluster structure for the data using K-Means and \( WCSS \) then generating cluster labels for the instances. The cluster labels are then used to train a DK to be used with kernel PCA to reduce the dimensionality of data to three. We further use the \( V_{3D} \) measurement to evaluate the quality of the dimension reduction process and determine if the data can be visualized in a 3-dimensional space with acceptable loss of cluster structure information.

Our experiments show that dimension reduction using unsupervised DK outperforms all other methods. Furthermore, kPCA using DK yields in no structural information loss. In other words, the cluster structure of the data is perfectly preserved in the 3-dimensional space.
Chapter 5

Deep Embedding Kernel

5.1 Introduction

The Deep Kernel (DK) architecture discussed in Chapter 3 is our first attempt in addressing the mapping mechanism in kernel methods and deep learning. The successes of DK are

- Providing kernel methods with a mechanism to have a kernel function trained from data instead of predefined.

- Representing a deep architecture that is able to map data to an implicit feature space. The training goal of the model in this case is also more generalizable as it is closer to that of kernel methods.

However, DK also has certain disadvantages. First, the Deep Belief Network (DBN) utilized in DK is relatively outdated and is usually difficult to train. Second, the data undergoes a non-linear transformation (into paired data) before being input into DK. This means the trainable parameters, or the hidden layers,
Chapter 5. Deep Embedding Kernel

of DK are not directly connected to the raw data which may limit its learning capabilities. There is also potential loss of information caused by the transformation. We illustrate the trainable parameters of DK in Figure 5.1. As can be seen, the pair of instances \((X^{(i)}, X^{(j)})\) is transformed into \(f(X^{(i)}, X^{(j)})\) before being input into the DK network. The trainable parameters only extends to \(f(X^{(i)}, X^{(j)})\) which may cause the problems discuss above. Furthermore, in practice, we have encountered situations where the training of DK cannot converge within a desirable number of iterations.

In this chapter, we discuss the Deep Embedding Kernel (DEK) architecture which is the core component of this dissertation. While addressing problems in kernel methods and deep learning like DK, DEK further solves problems of DK discussed above. In brief, the architecture of DEK integrates two learning networks, namely the **kernel network** and the **embedding network**. The kernel network outputs the pairwise similarity value for pair of data instances, while the embedding network feeds high-level representations of data into the kernel network. The training of both networks is done in a single gradient descent process with the same learning objective that specifies an optimized relationship...
of data in the desired feature space. We illustrate the two components of DEK in Figure 5.2 to show its differences compared to DK. The embedding network takes input as individual raw data instances, outputs their high-level representations \( U^{(i)} \) and \( U^{(j)} \), then feeds them into the kernel network to compute the pair’s similarity. As can be seen, the trainable parameters of DEK extends to the raw data (which is \( X^{(i)} \) and \( X^{(j)} \) without transformation). This design maximizes the learning capability of DEK and minimizes potential loss of information due to non-linear transformed input. We also discard the outdated DBN architecture and rely on other techniques to solve the gradient vanishing problem in deep learning.

![Trainable Parameters of Deep Embedding Kernel Architecture](image)

**Figure 5.2: The Trainable Parameters of Deep Embedding Kernel Architecture**

### 5.2 Methodology

The design goal of DEK is to learn an optimized feature space of data with desired features for the application. This optimized space is determined by DEK, a learnable kernel that is represented by a deep architecture. When we design
DEK, we consider the following factors. First, since it represents a kernel, DEK takes a pair of data instances as input and output their similarity. Similarity of data can be computed based on different representations of data at different abstraction levels. We want DEK to be able to learn data similarity based on optimized data representations. Then based on the given data representation, we want DEK to be able to learn a similarity function that is complex enough to map data to an optimized space with desired data distributions. Therefore, DEK is designed to have two learning components, namely embedding network and kernel network, integrated in a unified deep architecture. These two learning components will be trained using the same learning objective in a single learning process. The overall architecture of DEK is shown in Figure 5.3.

5.2.1 Kernel Network

The architecture of the kernel network is similar to that of DK without the uses of DBN. As shown in Figure 5.3, the input of the kernel network is denoted as $U^{(i,j)}$, which is formed by the outputs of the two branches of the embedding network, which are $U^{(i)}$ and $U^{(j)}$ respectively. More specifically, $U$ can be expressed as the following function of $U^{(i)}$ and $U^{(j)}$

$$U^{(i,j)} = \left\{ U_1^{(i)} \ast U_1^{(j)}, U_2^{(i)} \ast U_2^{(j)}, \ldots U_d^{(i)} \ast U_d^{(j)},\right.\left. |U_1^{(i)} - U_1^{(j)}|, \ldots |U_d^{(i)} - U_d^{(j)}| \right\}$$

(5.1)

Where $U_k^{(i)}$ denotes the $k^{th}$ dimension of $U^{(i)}$, and $d$ is the dimensionality of
5.2. Methodology

\[ K(X^{(i)}, X^{(j)}) \]

![Diagram of DEK](image)

**Figure 5.3: The Structure of DEK**

\( U^{(i)} \) and \( U^{(j)} \). In other words, each neuron in the input layer of the kernel network represents a symmetric relationship of \( U^{(i)} \) and \( U^{(j)} \) on a single dimension. The illustration of forming \( U^{(i)} \) from \( U^{(i)} \) and \( U^{(j)} \) is in Figure 5.4. In details, we concatenate the multiplication component (\( \{U^{(i1)} \cdot U^{(j1)}, \ldots U^{(id)} \cdot U^{(jd)}\} \)) and the absolute difference component (\( \{|U^{(i1)} - U^{(j1)}|, \ldots |U^{(id)} - U^{(jd)}|\} \)) into one input vector which is then fed into the kernel network.

The use of fine granularity of relationship on each individual dimension as input provides more room for learning, compared with directly using different predefined kernel functions on \( U^{(i)} \) and \( U^{(j)} \) as inputs. Essentially, this design
of inputs allows the kernel network to learn a kernel that is a highly non-linear combination of angles and distances of the data pairs in the space that is learned by the underneath embedding network. Furthermore, this design of inputs guarantees the output similarity is symmetric.

We use sigmoid as the output function for the kernel network. Therefore, the output of the kernel network can be interpreted as the probability that the two instances belongs to the same class. Intuitively, this probability can be viewed as a similarity value of the two instances; the higher the probability that the two instances belong to the same class, the more similar they should be. Formally,
5.2. Methodology

given instances \(i\) and \(j\), the output can be expressed as

\[
K(U^{(i)}, U^{(j)}) = P(y^{(i)} = y^{(j)}|U^{(i)}, U^{(j)}) = \text{sigmoid}(W_{out}^{(K)} \cdot H_{out}^{(K)} + b_{out}^{(K)}) \quad (5.2)
\]

With \(W_{out}^{(K)}\) and \(b_{out}^{(K)}\) being the parameters of the output layer, and \(H_{out}^{(K)}\) being the input of the kernel network. Because we have \(K(U^{(i)}, U^{(j)}) = K(U^{(j)}, U^{(i)}) > 0\), \(K(\cdot)\) is a kernel function.

To train the kernel network (as well as the whole DEK), we label each pair \((y^{(i)}, y^{(j)})\) as \(Y^{(i,j)}\), with

\[
\begin{cases}
Y^{(i,j)} = 1 \iff y^{(i)} = y^{(j)} \\
Y^{(i,j)} = 0 \iff y^{(i)} \neq y^{(j)}
\end{cases} \quad (5.3)
\]

That is, if instance \(i\) and \(j\) belong to the same class, the label for the pair of \(i\) and \(j\) is 1, otherwise it is 0. Then we define the learning objective of training DEK (including kernel network) is to minimize the following cost function.

\[
L = \sum_{data} (Y^{(i,j)} \log(K(U^{(i)}, U^{(j)}) + (1 - Y^{(i,j)}) \log(1 - K(U^{(i)}, U^{(j)})))) \quad (5.4)
\]

Minimizing \(L\) in equation (5.4) maximizes \(K(U^{(i)}, U^{(j)})\) if \(Y^{(i,j)} = 1\), and minimizes \(K(U^{(i)}, U^{(j)})\) if \(Y^{(i,j)} = 0\). In other words, through being trained with the cost function in (5.4), the kernel network learns to map the data to a feature space where the similarity of instances in the same classes is close to 1, while the similarity of instances in different classes is close to 0. Intuitively, the kernel network is trained to implicitly map data to a feature space where instances in
the same classes have high similarity and instances in different classes have low similarity.

5.2.2 Embedding Network

The purpose of the embedding network is to learn optimized high-level representations of data to feed into the kernel network as inputs. Let the mapping made by the embedding network be \( E(\cdot) \), the high-level representation of sample \( x^{(i)} \) can be represented as \( U^{(i)} = E(x^{(i)}) \). The goal of designing the embedding network is to increase the learning capacity of the final kernel. Experimental results demonstrate that the embedding network positively contributes to the performance of DEK.

The training of embedding network is in the same gradient descent process using the same cost function as in Equation (5.4).

5.2.3 Overall Design

Suppose the embedding network has \( k_1 \) hidden layers \( H_1^{(e)} \ldots H_{k_1}^{(e)} \) and the kernel network has \( k_2 \) hidden layers \( H_1^{(K)} \ldots H_{k_2}^{(K)} \). Also suppose the input layer of the embedding network is \( H_0^{(e)} \) and of the kernel network is \( H_0^{(K)} \), and the weights and bias of layer \( i \) of network \( j \) are \( W_i^{(j)} \) and \( b_i^{(j)} \). The computational flow from a pair of instances \( (X^{(i)}, X^{(j)}) \) can be expressed as

- The embedding of \( X^{(i)} \):
  \[
  H_0^{(e)}(i) = X^{(i)}
  \]
  \[
  H_1^{(e)}(i) = \sigma(W_0^{(e)} \cdot H_0^{(e)}(i) + b_0^{(e)})
  \]
5.2. Methodology

\[ U^{(i)} = H_{k_1}^{(e)}(i) = \sigma(W_{k_1-1}^{(e)} \cdot H_{k_1-1}^{(e)}(i) + b_{k_1-1}^{(e)}) \]

- The embedding of \( X^{(j)} \):
  \[ H_0^{(e)}(j) = X^{(j)} \]
  \[ H_1^{(e)}(j) = \sigma(W_0^{(e)} \cdot H_0^{(e)}(j) + b_0^{(e)}) \]

\[ U^{(j)} = H_{k_1}^{(e)}(j) = \sigma(W_{k_1-1}^{(e)} \cdot H_{k_1-1}^{(e)}(j) + b_{k_1-1}^{(e)}) \]

- Input to the kernel network:
  \[ U^{(i,j)} = U^{(i)} \bullet U^{(j)} = H_0^{(K)} \]
  \[ H_1^{(K)} = \sigma(W_0^{(K)} \cdot H_0^{(K)} + b_0^{(K)}) \]

\[ H_{k_2}^{(K)} = \sigma(W_{k_2-1}^{(K)} \cdot H_{k_2-1}^{(K)} + b_{k_2-1}^{(K)}) \]
  \[ K(x^{(i)}, x^{(j)}) = s(W_{k_2}^{(K)} \cdot H_{k_2}^{(K)} + b_{k_2}^{(K)}) \]

with \( \sigma(\cdot) \) being the activation function, \( s(\cdot) \) being the output function, and

"\( \bullet \)" being the dimension-wise similarity operator as discussed:

\[ U^{(i,j)} = \begin{cases} U^{(i_1)} * U^{(j_1)}, & U^{(i_2)} * U^{(j_2)}, \ldots U^{(i_d)} * U^{(j_d)}, \\ \left| U^{(i_1)} - U^{(j_1)} \right|, & \ldots \left| U^{(i_d)} - U^{(j_d)} \right| \end{cases} \]
Layers in both component network are updated with gradient descent:

\[
W_i^{(j)} \leftarrow W_i^{(j)} - \alpha \frac{\partial L}{\partial W_i^{(j)}} \\
b_i^{(j)} \leftarrow b_i^{(j)} - \alpha \frac{\partial L}{\partial b_i^{(j)}}
\] (5.5)

A unified structure is currently being employed on all layers to simplify the training process. In detail, all embedding layers have \( k \) hidden neurons, and all kernel layers have \( 2k \) neurons, where \( k = \alpha d \) with \( d \) being the dimensionality of the original data and \( \alpha \) being an integer factor (typically, we use \( \alpha \in \{1, 2, 3, 4\} \)).

### 5.3 Training the Deep Embedding Kernel

In this section, we discuss how DEK can be trained in supervised learning for different tasks. We primarily focus our discussion on the following supervised learning tasks: identity detection, general classification, and dimension reduction for visualization.

The problem of identity detection can be defined as assigning an identity to a query sample (e.g., a facial image). A common supervised learning strategy to solve this problem is to assign an identity to the query sample based on its nearest neighbors in the training set.

Identity detection with DEK feeds the query sample and each of the training samples into the trained deep network and finds the nearest neighbors of the query sample using the outputted kernel values. The learning objective of DEK, which set the desired similarity of two samples of the same identity to be 1 and the desired similarity of two samples of different identities to be 0, naturally fits
into the problem of identity detection. Therefore, any pair from the training data can be used to tune the network of DEK for identity detection.

The problem of general classification can be defined as assigning a class label to a query sample. Given that two samples belong to the same class may not necessarily have the same level of similarity as two samples of the same identity, it may be over-strict for the learning objective of DEK to set the desired similarity of two samples from the same class to be 1. Therefore, a local pairing strategy is proposed to generate training pairs to tune a DEK for a classification problem.

The local pairing strategy works as follows. First, all pairs of data are fed into DEK; each sample is used as reference to rank all other samples in descending order of kernel values outputted by DEK. A certain recall level (e.g., 0.1) is then used to determine the neighborhood of the reference sample. Within the neighborhood, we form positive pairs between the reference sample and the samples of the same class, and form negative pairs between the reference sample and the samples of different classes. The local pairing strategy is illustrated in Figure 5.5. By using local pairing strategy, we avoid to force the similarity of distant samples of the same class to be close to 1. In order to improve the efficiency of the training, the local pairing strategy is applied to generate training pairs at certain interval of iterations. For example, local pairing strategy is applied to generate training pairs at the 1st, 51st, 101st, 151st, ... iterations. Other iterations between the interval use the most-recently generated training pairs.
5.4 Experiment Study

In this section, we describe applications of DEK on general classification and data visualization. We also present an experiment to show the contribution of the embedding network component in DEK. The application of DEK in identity detection is discussed in more detail in Chapter 6. All experiments for each of the above tasks are conducted in Python version 2.7.12. Deep models are implemented using the package Theano (Bergstra et al., 2010), other machine learning models (including the regular DNN) are from the Sci-Kit Learn (Pedregosa et al., 2011) package. Visualizations are generated using the Matplotlib library (Hunter, 2007).

5.4.1 General Classification

To study the performance of DEK with local pairing strategy on general classification, we compare SVM using DEK (S-DEK) and KNN using DEK (K-DEK)
with other classification models including SVM using RBF kernel (R-SVM), Gradient Boosting Trees (GB) (Friedman, 2002), Random Forest (RF) (Liaw and Wiener, 2002), and DNN on seven datasets.

We perform experiments on seven datasets:

1. Messidor Features (Decencière et al., 2014). The data contains 18 features extracted from 1151 eye images. The task is to classify if the eye has diabetic retinopathy presences or not.

2. Pima Indian Diabete (Smith et al., 1988). The data was collected from the Pima Indian population near Phoenix, Arizona. Eight dimensions of the data include number of times pregnant, plasma glucose concentration from an oral glucose tolerance test, diastolic blood pressure, triceps skin fold thickness, 2-hour serum insulin, body mass index, diabetes pedigree function, and age. The class label is whether the patients had diabetes or not. The dataset has 768 instances.

3. Segment (Zhang, 1992). The data consists of 2310 instances with 19 attributes. Each instance was originally an outdoor images of which features were manually extracted by the researchers. There are seven classes in the data.

4. Cardiotocography (Campos et al., 2000). The data consists of 23 features extracted from 2126 cardiotocograms. The data was categorized into 10 classes by three expert obstetricians.

5. Seismic Bumps (Sikora and Wróbel, 2010). The data records the energy of seismic bumps in a coal mine in Polish. There are 19 attributes and 2584
instances in the data. The goal is to determine if the instance is hazardous or non-hazardous.

6. QSAR biodegradation (Mansouri et al., 2013). The data consists of 41 attributes (molecular descriptors) of 1055 chemicals. The goal is to determine if the chemical is ready biodegradable or not ready biodegradable.

7. Yeast (Horton and Nakai, 1996). This dataset consists of 1484 instances with 8 attributes. The goal is to predict the Cellular Localization Sites of each instance.

Each dataset is split into 70% training data, 15% validation data, and 15% testing data.

In all experiments, we use a DEK architecture of two embedding layers and three kernel layers (including output layer). All embedding layers have $d$ neurons, and all kernel layers have $2d$ neurons, with $d$ being the dimensionality of the input data. As DEK does not directly make decisions, we monitor the training of DEK according to the change of training cost value and terminate the training when the decrease in training cost is significantly slowing down in adjacent epochs. We start training with a learning rate of 0.1 which is then decreased by a factor of 10 whenever the training cost heavily fluctuates among consecutive epochs. We use a recall level of 0.5 for our local pairing strategy to form training pairs in all experiments.

After DEK is trained to convergence, we use the trained DEK in two kernel machines as mentioned before, namely, SVM and KNN. We use grid-search


5.4. Experiment Study

Table 5.1: Test Accuracy Rates of Models in Experiments for Classification

<table>
<thead>
<tr>
<th>Dataset</th>
<th>S-DEK</th>
<th>K-DEK</th>
<th>SVM</th>
<th>GB</th>
<th>RF</th>
<th>DNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messidor Features (Decencièr et al., 2014)</td>
<td>78.03</td>
<td>77.46</td>
<td>69.36</td>
<td>68.79</td>
<td>69.94</td>
<td>75.14</td>
</tr>
<tr>
<td>Pima Diabete (Smith et al., 1988)</td>
<td>77.57</td>
<td>76.72</td>
<td>72.41</td>
<td>72.41</td>
<td>76.72</td>
<td>68.97</td>
</tr>
<tr>
<td>Segment (Zhang, 1992)</td>
<td>97.41</td>
<td>98.27</td>
<td>96.83</td>
<td>97.98</td>
<td>98.27</td>
<td>97.41</td>
</tr>
<tr>
<td>Cardiotocography (Campos et al., 2000)</td>
<td>99.06</td>
<td>99.37</td>
<td>99.06</td>
<td>99.06</td>
<td>99.37</td>
<td>99.37</td>
</tr>
<tr>
<td>Seismic Bumps (Sikora and Wróbel, 2010)</td>
<td>94.07</td>
<td>93.30</td>
<td>94.07</td>
<td>93.81</td>
<td>93.81</td>
<td>94.07</td>
</tr>
<tr>
<td>QSAR biodegradation (Mansouri et al., 2013)</td>
<td>86.16</td>
<td>83.02</td>
<td>85.53</td>
<td>83.64</td>
<td>83.02</td>
<td>85.53</td>
</tr>
<tr>
<td>Yeast (Horton and Nakai, 1996)</td>
<td>51.57</td>
<td>58.74</td>
<td>52.91</td>
<td>55.16</td>
<td>55.16</td>
<td>51.57</td>
</tr>
</tbody>
</table>

with validation accuracy to optimize the penalty hyper-parameter in SVM. In all experiments, we use 25-NN (i.e. KNN with \( K = 25 \)).

The processes of optimizing hyper-parameters that are used in control models are described as follows. The hyper-parameters used by R-SVM are optimized via grid-search using validation accuracy. For GB and RF, we train 10 models for each of them on training data and select the one with the highest validation accuracy to apply on the testing set. Last, for the control model using DNN, we use validation accuracy to choose the model with the optimal number of hidden layers.

The test performances of all models are presented in Table 5.1.

As can be seen, either DEK-based SVM or DEK-based KNN achieve the best results in all datasets (including three cases in which DEK based models have equal-best with one of the control models).
5.4.2 Data Visualization

In this experiment, we train DEK with three datasets: Segment, Cardiotocography, and Waveform (Breiman et al., 1984). The trained DEK’s are then utilized as kernel functions for kernel Principal Component Analysis (kPCA) to reduce the datasets’ dimensionality to three and visualize them. We compare DEK to the commonly used RBF kernel (also in combination with kPCA) in this task.

Figure 5.6 shows the visualization with DEK/kPCA and RBF/kPCA of the three datasets. As can be seen, DEK map data to a feature space where instances in the same class are in a more compressed areas compared to the RBF kernel. Visibly, the classes are also more linearly separable in all three cases. This shows the superiority of DEK over the traditional RBF kernel in dimension reduction and visualization.

5.4.3 The Role of the Embedding Network

In the last experiment, we want to show the importance of the embedding network component to DEK. Theoretically, a deep enough network can represent any function between the input and output; in reality, however, the practical training algorithm - usually gradient descent - may not be able to guarantee that this function is learnable (Goodfellow, Bengio, and Courville, 2016). Therefore, in the situation where the function that describes an optimal pairwise similarity relationship of the data based on the original data representation is complex enough, the kernel network component of DEK by itself may not be able to learn this function (practically, the training process may not converge). However, if we can first transform the data from its original representation to an high-level
5.4. Experiment Study

Figure 5.6: Visualization with Kernel PCA using Deep Embedding Kernel and RBF Kernel
representation that is optimized towards the pairwise similarity calculation, we may be able to reduce the complexity of such a function based on the transformed data representation. Moreover, since the input of the kernel network is a vector of dimension-wise similarities that are transformed from the pair of input instances, instead of the raw instances themselves, the learning process may lead to information loss. The embedding network solves the two problems by not only learning a high-level data representation that is optimized for pairwise similarity calculation thus reducing the complexity of the function that the kernel network component needs to learn, but also extending the learning to the raw data level to avoid any potential information loss. In our experiments, a standalone kernel network of over two hidden layers may not be able to converge in certain datasets while adding even one embedding layer to form DEK yields a trainable network.

For experiment, we use embedding vectors of the Indian Movie Face Database (IMFDB) (Setty et al., 2013) that are generated using Google Facenet (Schroff, Kalenichenko, and Philbin, 2015) as training and testing data for DEK and DK. In brief, the IMFDB data consists of facial images of Indian actors and actresses that were extracted from movies. The facial images are fed into Google Facenet to transform them into 300-dimensional vectors which are used as input to the networks in this experiment. We discuss this architecture in more details in Chapter 6. Both DEK and DK are trained independently on IMFDB. The DEK model has two embedding layers and two kernel layers; the DK model has four hidden layers. The precision-recall curves of both models are plotted in Figure 5.7. It can be seen that the embedding network of DEK contributes significantly
towards the performance. This experimental result supports our hypothesis that the incorporating of the embedding network in DEK increases the learning capacity of the model.

![Figure 5.7: Contributions of Embedding Layers to DEK Performance](image)

**Figure 5.7:** Contributions of Embedding Layers to DEK Performance

## 5.5 Conclusion

In this chapter, we propose the Deep Embedding Kernel architecture to automatically learn an optimized feature space from training data. DEK is represented by a deep neural network that consists of two components: a deep embedding network and a deep kernel network. The integration of these two components in a unified framework maximizes the learning power of the deep architecture. The deep embedding network is designed to learn high-level representations, while the deep kernel network is designed to further learn non-linear similarities. We show that DEK outperforms commonly-used machine learning models in general classification and dimension reduction.
Chapter 6

The Deep Embedding Kernel - Dual Deep Learning Framework

6.1 Introduction

Deep Embedding Kernel (DEK) can be directly applied to structured records for supervised learning. For other types of data, such as image data or sequential data, DEK can be laid on top of those deep architectures that are able to output vector embedding for the corresponding data, such as CNNs on image data and RNNs on sequential data. Here, we consider the output of the last hidden layer of CNNs or RNNs as vector embedding to be input into DEK. For example in image classification, one can remove the SoftMax output layer of a CNN then fully connect the CNN to a DEK to form the new architecture. The deep neural network with DEK on top for both image data and sequential data are shown in Figure 6.1.

However, like other deep learning models, the challenge emerges with big
datasets. Due to the complexity of a deep neural network (DNN) and the iterative nature of Gradient Descent, training DNN’s is usually considerably longer than other machine learning models and requires the use of powerful Graphic Processing Units (GPUs). Even then, training DNN’s in big datasets might still be costly for the general users. For example, Google Facenet (Schroff, Kalenichenko, and Philbin, 2015) was trained for about two weeks on a set of two million facial images using Google GPU cluster.

Moreover, while users can utilize pre-trained version of complex deep networks, the models’ performance might drop when applied to datasets with different characteristics than their original training data. In Figure 6.2, we show the
6.1. Introduction

precision-recall (PR) curve measures Google Facenet performance when applied to the Indian Movie Face Database (IMFDB) (Setty et al., 2013). The IMFDB data consists of facial images of Indian actors and actresses extracted from movies, and can be considered vastly different from the original training data of Google Facenet. As can be seen, the PR curve of Facenet on the IMFDB data is not ideal and can be improved. The cost to continue to train models like Facenet in new data might become prohibitive for users, as discussed above.

![Figure 6.2: Precision-Recall Curve of Google Facenet in the IMFDB Data](image)

These two issues of being computationally expensive and not generalizable to new data with new characteristics motivated us to design the Dual Deep Learning (DDL) framework that allows users to effectively model big data with deep models. The DDL framework is the results of our work in collaboration with Equifax Inc., and has been filed as a US Patent in 2018 (Xie and Le, 2018). In short, the DDL framework consists of two DNN’s trained in two phases, data representation learning and data relationship learning. Phase I aims to train a
DNN that can provide a good data representation for a data type, or a general problem. In the first phase, any network that is suitable for the data type or the problem can be used. Alternatively, pre-trained networks can be utilized in place of phase I network to reduce training cost and make use of public knowledge. Phase II network is then trained on top of the phase I network towards the given task. This design provides users with a unified framework to model big data with deep networks. A pre-trained complex network like Google Facenet can also be easily adapted to new data.

More recently, we integrate our DEK architecture into phase II of the DDL framework to form a unified DEK/DDL framework for big unstructured data. In this chapter, we discuss our DEK/DDL framework, as well as showcase an application of the framework in facial recognition.

6.2 The Deep Embedding Kernel - Dual Deep Learning Framework

The training process and overall architecture of DEK/DDL framework is shown in Figure 6.3. In short, training the DEK/DDL framework consists of two phases which are data representation learning and data relationship learning. The output of phase I network is embedding vectors for individual data instances. DEK is integrated as the phase II network to output similarities for pairs of instances based on their embedding vectors from phase I network. In decision making, a kernel machine using DEK as its kernel function can be utilized.
In the first learning phase, the DNN is trained to generate a good data representation for the type of data associated to the given task. This means the training in this phase is not limited to the user data, but can be done in any external data source having the same characteristics. For example, in a facial recognition task, the phase I model is trained either with the user data or any large
public facial data set like Labeled Face in the Wild (LFW) (Huang et al., 2008). Depending on the type of data, various types of deep models can be used in this phase, e.g. Multilayer Perceptron (MLP) for structured data, Convolutional Neural Network (CNN) for image data, Recurrent Neural Network (RNN) for sequential data, etc. Alternatively, a pretrained deep network may be utilized, e.g. Google FaceNet is a good option as phase I model for any facial recognition task. Overall, the goal of this phase is to establish a general knowledge base for the data before modeling, represented by the embedding space mapped from the trained deep network. Given data labels are available in the phase I data, deep networks using the Triplet Loss (Schroff, Kalenichenko, and Philbin, 2015) cost function are good candidates. Otherwise, unsupervised deep networks like Deep Belief Network (Wang and Raj, 2015) or Stacked Auto-Encoder (Vincent et al., 2010) can be used.

After phase I learning, DEK is put on top of the trained phase I network to learn the data relationship that is optimal for the given task. The training data in this phase is the user data, and DEK is trained with a loss function that is tied to the given task. For example, to perform facial recognition on the IMFDB data, one can utilize a DEK/Facenet framework as follows

- Phase I network is Google Facenet. The user can get a public pretrained version to leverage the lack of computational resources.

- Phase II network is DEK. DEK takes inputs as embedding vectors output by phase I Facenet, and is trained using the IMFDB data. The final goal is facial recognition, so DEK is trained to maximize similarities of facial
images from the same persons, and minimize similarities of facial images from different person.

- A K Nearest Neighbors (KNN) approach can be used to make the decision. Given a facial image with unknown identity, the similarities of the unknown image and the training images are first obtained. We then rank the training images using the computed similarities and predict the identity of the unknown image based on its nearest neighbors.

The architecture of the DEK/Facenet framework for facial recognition is shown in Figure 6.4.

**Figure 6.4:** The Deep Embedding Kernel - Dual Deep Learning Framework for Facial Recognition

In the next section, we formally define the problem of facial recognition and discuss the example of DEK/Facenet framework in details.
6.3 Experimental Study in Facial Recognition

The problem of facial recognition can be defined as assigning an identity to a query sample which is a facial image. A common supervised learning strategy to solve this problem is to assign the identity to the query sample based on its nearest neighbors in the training set.

For this task, we utilize the DEK/DDL framework as described in Section 6.2 with Google Facenet as the phase I network. In this experiment, we also want to demonstrate the use of DEK in transfer learning. Therefore, we choose a pretrained Google Facenet (Schroff, Kalenichenko, and Philbin, 2015) (available from https://github.com/davidsandberg/facenet) instead of training Facenet in our given data.

In brief, Google Facenet is a deep CNN trained with the Triplet Loss cost function as discussed in Chapter 2. The output of Facenet is a 300-D embedding vector. We use these output vectors as input to DEK. To train DEK as a transfer learning model, we first feed the new training data into Google FaceNet to obtain their embedding vectors. We then train the DEK component using the newly generated embedding vectors as training data for DEK. The Google FaceNet component is kept unchanged.

After DEK is fully trained, we can make decision in the process described as follows:

1. Feeding facial image into FaceNet to obtain their vector embedding.
2. Feeding the vector embedding into the embedding network to obtain their higher-level embedding.
3. Feeding pairs of higher-level embedding into the kernel network to obtain their pairwise similarity as the final output.

4. Using a K-Nearest-Neighbor approach to assign identities to unknown data as discussed in Section 6.2

In this experiment, the data we use is Indian Movie Face Database (IMFDB) – (Setty et al., 2013). This dataset contains facial images of Indian movie actors and actresses. The DEK component that is laid on top of Google FaceNet has two embedding layers and two kernel layers, and is trained on the vector embedding of IMFDB generated by FaceNet. To evaluate the performance of DEK, we also build another transfer learning model as the control model based on the pretrained Facenet and compare it with the Facenet/DEK model. The control model lays a 4-layer deep neural network on top of the pretrained FaceNet. The top deep neural network is trained on IMFDB using Triplet Loss. We denote this control model as DNN/TL. Both models are trained and tested on the same subsets from the IMFDB data (75% training, 25% testing). The trained DNN/TL outputs vector embedding based on which we can compute the pairwise distances among images for identity assignment.

To evaluate the two models, each image in the testing set is used as a query image to rank all images in the training set in the ascending order of their distances outputted by the DNN/TL model, and in the descending order of similarities outputted by DEK. We then plot the average precision-recall curve for these two rankings. We also plot the precision-recall curve generated by the pretrained FaceNet without transfer learning as the baseline. As shown in Figure 6.5, both transfer learning models make substantial improvements over the
pretrained FaceNet. DEK makes further improvement over DNN/TL at almost every recall level. Given DNN/TL has already achieved near-perfect precisions, the further improvement made by DEK is significant. Therefore, DEK can be used as the desired solution to facial recognition in critical applications where very high accuracy is demanded.

**Figure 6.5:** Performance of Transfer Learning Models with DEK and DNN/TL on Facial Recognition
Chapter 7

Deep Similarity-Enhanced K Nearest Neighbors

7.1 Introduction

K Nearest Neighbors (KNN) (Peterson, 2009) is a simple yet effective algorithm in statistics and machine learning. Given a distance or similarity metric, KNN first retrieves the $k$ nearest neighbors of the target instance, then assigns the class label that has the highest frequency in $k$ nearest neighbors to the target instance. While KNN is simple, it is widely applied in numerous analytical tasks, including, but not limited to


- Object recognition (Bajramovic et al., 2006)
• Web mining (Adeniyi, Wei, and Yongquan, 2016)

• Medical applications (Korn et al., 1998) (Deekshatulu and Chandra, 2013).

Furthermore, the decisions made by KNN are typically intuitive and easily understandable, especially with some visualizations of neighborhoods. In (Xie, 2016), the authors further demonstrated that KNN can be used to integrate heterogeneous features for decision making.

Since KNN makes decisions based on neighborhoods of data instances, its performance heavily depends on the quality of the neighborhoods. If the neighborhoods in the original space of data are mixed with instances from different classes, KNN is likely to yield poor classification accuracy. A typical solution to this issue is to map the data to a different feature space where the quality of the neighborhood may be improved. The works in (Globerson and Roweis, 2006) (Goldberger et al., 2005) (Weinberger and Saul, 2009) (Xing et al., 2003) use certain linear transformations that are optimized for KNN decision making to map data to a feature space. However, these linear transformations may not be able to capture the nonlinear correlation between class labels and data features. Kernel KNN (Zuo, Zhang, and Wang, 2008) uses a predefined/preselected kernel function to implicitly map data to a feature space. However, there are no guarantees that the mapped feature space is optimal towards KNN decision making (Le and Xie, 2019).

Deep learning, with many recent breakthrough successes, has also been applied to improve KNN. The works in (Min et al., 2009) and (Ren et al., 2014) use deep neural networks (DNN) to learn nonlinear transformations that map data
to a feature space with predefined dimensions, where the quality of the neighborhood is optimized. The DNNs used in (Min et al., 2009) include Stacked Auto-Encoder (Vincent et al., 2010) and stacked Restricted Boltzmann Machine (Hinton, 2009). The learning goal of the DNN in (Min et al., 2009) is that the $k$ nearest neighbors of a reference instance are all from the same class as the reference instance itself, and the nearest instance from a different class is at least a margin away from the $k$ nearest neighbors. The work in (Ren et al., 2014) uses Convolutional Neural Network (CNN) (Krizhevsky, Sutskever, and Hinton, 2012) to transform image data to a feature space with predefined dimensions with the goal that the expected number of instances that are correctly classified by KNN is maximized. However, the similarity/distance functions used in these works, such as Euclidean distance or Mahalanobis distance, have to be tied with a feature space of predefined dimensions. We think this limitation on the types of similarity/distance functions that can be used actually limit the learning capacity of these methods.

Instead of learning an optimized feature space with predefined dimensions (which intrinsically restricts distance/similarity functions that can be used for KNN), in this chapter, we propose a novel deep architecture, called the Deep Similarity-Enhanced K Nearest Neighbors (DSE-KNN), to automatically learn an optimized similarity function of the data directly towards the goal of optimizing KNN decision making. In other words, the deep architecture of DSE-KNN learns to implicitly map the data to high-dimensional feature space where the accuracy of KNN decision making is maximized.

In brief, the architecture of DSE-KNN uses the Deep Embedding Kernel (DEK)
as its core component, and a Similarity-Enhanced KNN (SE-KNN) layer on top of DEK. The SE-KNN is the decision component of the model. As discussed in Chapter 5, the kernel network (KN) of DEK is to output pairwise similarity that is optimized for SE-KNN, and the embedding network (EN) of DEK is to output the non-linear embedding of input data that is optimized for KN. Both KN and EN will be trained in an integral training process via back-propagation towards the goal of maximizing the decision accuracy of SE-KNN. Maximizing the decision accuracy of SE-KNN is quantified as an equivalent process of minimizing the KNN loss - a new loss function that is proposed in this chapter. Our experimental studies show that DSE-KNN outperforms various machine learning methods, including deep learning approaches on classification on different types of disease data.

7.2 Methodology

DSE-KNN is a deep architecture that consists of three components: an embedding network (EN), a kernel network (KN), and a similarity-enhanced KNN (SE-KNN) functioning as the decision making model (recall that the EN and KN component form the DEK architecture that is discussed in Chapter 5). DSE-KNN network is trained using KNN loss to optimize the accuracy of the SE-KNN decision making.

A trained DSE-KNN makes decisions for a unknown instances using a process that is shown in Fig. 7.1. First, each instance $X^{(i)}$ in the training dataset
and the unknown instance $X^{(s)}$ are fed into the EN to generate their embedding vectors $U^{(i)}$ and $U^{(*)}$. These vectors are then input into the KN to generate the pairwise similarities between each training instance $X^{(i)}$ and the unknown instance $X^{(*)}$, denoted as $s(X^{(i)}, X^{(*)})$. The computed similarities are used to determine the nearest neighbors, denoted as $KNN^{(*)}$, of the unknown instance $X^{(*)}$ from the training data. Finally the label $Y^{(*)}$ of $X^{(*)}$ is determined based on $KNN^{(*)}$ using SE-KNN.

![Figure 7.1: The Decision Making Process of DSE-KNN](image)

The training of the DSE-KNN is directly towards the goal of maximizing
Chapter 7. Deep Similarity-Enhanced K Nearest Neighbors

KNN decision accuracy. In other words, DSE-KNN learns a similarity function that is optimized for KNN decision accuracy. In order to quantify KNN decision accuracy, we use a complementary measure that is called KNN loss. Therefore, maximizing KNN decision accuracy can be performed by minimizing the KNN loss.

7.2.1 Similarity-Enhanced KNN

We use a modified version of KNN that is called similarity-enhanced KNN (SE-KNN) to have the decision accuracy directly correlated to the learning goal of DSE-KNN, i.e., training DSE-KNN with KNN loss directly improves the accuracy of similarity-enhanced KNN. The process of SE-KNN decision making is described as follows.

Let $X^{(i)}$ and $X^{(j)}$ be two data instances with label $y^{(i)}$ and $y^{(j)}$. Further assume $K(\cdot)$ is a function that computes the pairwise similarity among data in a feature space $S$, i.e., the similarity of $X^{(i)}$ and $X^{(j)}$ in $S$ can be denoted as $K(X^{(i)}, X^{(j)})$. Therefore, in $S$, $X^{(i)}$ and $X^{(j)}$ are similar if $K(X^{(i)}, X^{(j)})$ is large, and $X^{(i)}$ and $X^{(j)}$ are dissimilar if $K(X^{(i)}, X^{(j)})$ is small.

We can then define the neighborhood $KNN^{(i)}$ of $X^{(i)}$ in the feature space $S$ as the set of $k$ instances which have the highest similarity to $X^{(i)}$. In this neighborhood, the probability of $y^{(i)}$ being class $c_l$ among the classes $\{c_1, c_2, ..., c_n\}$ is defined as

$$P(y^{(i)} = c_l) = \frac{\sum_{y^{(j)} = c_l \in KNN^{(i)}} K(X^{(i)}, X^{(j)})}{\sum_{j \in KNN^{(i)}} K(X^{(i)}, X^{(j)})}$$

(7.1)
7.2. Methodology

In details, the probability of \( y^{(i)} \) being \( c_l \) is computed by the aggregated similarity between \( X^{(i)} \) and all instances with class \( c_l \) in \( KNN^{(i)} \) divided by the sum of similarities between \( X^{(i)} \) and all instances in \( KNN^{(i)} \). \( X^{(i)} \) can then be assigned to the class with the highest probability. Therefore, by SE-KNN, the label of an instance is determined not by the frequency of the other instances’ labels in the neighborhood, but by the aggregated similarity of each class in the neighborhood. Figure 7.2 illustrates the \( KNN^{(i)} \) for \( X^{(i)} \) and the calculation of the probability that \( X^{(i)} \) belongs to \( c_1 \) and \( c_2 \), respectively.

\[
p(y^{(i)} = c_1) = \frac{\sum_{y^{(j)} = c_1} s(X^{(i)}, X^{(j)})}{\sum_{j} s(X^{(i)}, X^{(j)})}
\]

\[
p(y^{(i)} = c_2) = \frac{\sum_{y^{(k)} = c_2} s(X^{(i)}, X^{(k)})}{\sum_{k} s(X^{(i)}, X^{(k)})}
\]

**Figure 7.2**: The Decision Making Process of SE-KNN

7.2.2 KNN Loss

The KNN loss described in this section directly correlates with the decision accuracy of the SE-KNN presented in Section 7.2.1. The KNN loss is calculated over all instances in the training data. We first define the training neighborhood neighborhood \( tKNN^{(i)} \) of \( X^{(i)} \) in the feature space \( S \) as the set of instances which
have the highest similarity to \( X^{(i)} \) and has exactly \( k \) instances of the same label with \( X^{(i)} \) (in implementation, we query all instances until the \((k + 1)th\) neighbor of the same label as \( X^{(i)} \)). Mathematically:

\[
tKNN^{(i)} = \{X^{(j)} : K(X^{(i)}, X^{(j)}) \geq s(X^{(i)}, X^{(k+1)}) \}
\] (7.2)

with \( X^{(k+1)}_+ \) being the \((k + 1)th\) neighbor having the same label as \( X^{(i)} \). Then, to compute the portion of KNN loss for \( X^{(i)} \), we pair \( X^{(i)} \) with each training instance in its training neighborhood \( tKNN^{(i)} \), and label the pair \((X^{(i)}, X^{(j)})\) as \( Y^{(i,j)} \) using the criteria specified in (7.3),

\[
\begin{aligned}
Y^{(i,j)} = 1 & \iff y^{(i)} = y^{(j)} \\
Y^{(i,j)} = 0 & \iff y^{(i)} \neq y^{(j)}
\end{aligned}
\] (7.3)

The KNN Loss in the K-nearest neighborhood of \( X^{(i)} \) can then be mathematically expressed as the Binary Cross-Entropy of the paired data in its training neighborhood:

\[
L^{(i)} = - \sum_{j \in tKNN^{(i)}} (Y^{(i,j)} \log(K(X^{(i)}, X^{(j)}) + (1 - Y^{(i,j)}) \log(1 - K(X^{(i)}, X^{(j)})))
\] (7.4)

According to equation (7.4), within the neighborhood of \( X^{(i)} \), the similarities between \( X^{(i)} \) and neighbors of the same class negatively contribute to the loss, while the similarities between \( X^{(i)} \) and neighbors of different class positively contribute to the loss.
The KNN loss over the training data is then the sum of losses from each training instance as shown in (7.5)

\[ L = \sum_{i \in \text{training data}} L^{(i)} \] (7.5)

As can be seen from the definition of KNN loss, lowering the KNN loss directly leads to improving the SE-KNN decision accuracy. Therefore, we use the KNN loss function for training DSE-KNN, such that the similarity function represented by DSE-KNN will learn to map data to a high dimensional feature space where KNN loss is minimized (equivalently, decision accuracy is maximized).

### 7.2.3 DSE-KNN Architecture

As discussed, the SE-KNN model described in Section 7.2.1 acts as the decision layer for DSE-KNN. The similarity function that is utilized in SE-KNN is represented through a DEK network. DEK consists of an embedding network (EN) and a kernel network (KN), both of which are optimized through the KNN Loss function described in Section 7.2.2. Overall, DSE-KNN is trained to map data to a feature space where the neighborhood of each instance is optimized for the decision making of SE-KNN.

The training process of DSE-KNN on a training set can be described as follows.

1. Given a training dataset \( X \), the whole training set is input into the EN that outputs the embedding for each instance ("Generating Embedding" step in
Figure 7.3).

2. Every pair of embedding vectors are fed into the KN that outputs its pairwise similarity ("Computing Pairwise Similarity" step in Figure 7.3).

3. For each training instance $X^{(i)}$, its training neighborhood $tKNN^{(i)}$ is determined from the training set as discussed in Section 7.2.2) ("Forming $tKNN_i$" step in Fig. 7.3).

4. Compute the KNN loss of each $X^{(i)}$ using (7.4) ("KNN Loss for $X^{(i)}$" step in Fig. 7.3)

5. The KNN loss for each training instance is aggregated to the overall KNN loss using equation (7.5) ("Aggregating KNN Loss" step in Fig. 7.3).

6. Use Gradient Descent to update the weight matrices and bias vectors throughout the DSE-KNN network in order to minimize the aggregated KNN loss (The backward arrows in Fig. 7.3).

Our experiments show that $k$ ranges from 25 to 50 usually yields good model performance. Moreover, we observe that ranking and selecting $k$ nearest neighbors for all instances in every epoch results in training instabilities and slower convergence speeds. Therefore, we regenerate training pairs after a certain number of epochs instead (e.g. 10, 50, or 100 epochs).
7.2. Methodology

![Diagram](image)

**Figure 7.3:** Illustration of the Training Process using KNN Loss $t$
7.3 Experimental Study: Classification on Medical Data

We implement all experiments in Python 2.7.15rc1. All deep models are implemented using Theano (Bergstra et al., 2010). Other machine learning models are available in the Scikit-Learn package (Pedregosa et al., 2011).

Given that disease classification/prediction is a critical application of supervised learning, in this experimental study, we compare the performance of DSE-KNN with other mainstream classification methods, including SVM using RBF kernel (SVM/RBF), Gradient Boosting Trees (GB) (Friedman, 2002), Random Forest (RF) (Liaw and Wiener, 2002), and Deep Neural Network on several disease datasets.

The datasets we use in this experiment study are described as follows.

1. Pima Diabetes (Smith et al., 1988) dataset. The data was collected from the Pima Indian population near Phoenix, Arizona. Eight dimensions of the data include number of times pregnant, Plasma Glucose Concentration from an Oral Glucose Tolerance Test, Diastolic blood pressure, triceps skin fold thickness, 2-hour serum insulin, body mass index, diabetes pedigree function, and age. The class label is whether the patients had diabetes or not. The dataset has 768 instances.

2. Diabetic Retinopathy Debrecen (Decencière et al., 2014) dataset. The data contains 18 features extracted from 1151 eye images. The task is to classify if the eye has diabetic retinopathy presences or not.
3. Breast Cancer Wisconsin (Wolberg and Mangasarian, 1990). The data was obtained from clinical cases in the University of Wisconsin Hospitals, Madison. The dataset has 699 cases with ten features regarding the patients’ diagnostic information. The goal is to determine if a case is benign or malignant.

4. Autism Screening (Thabtah, 2017). The dataset is used for classification of Autistic Spectrum Disorder (ASD) among 704 cases. The data has 20 features including demographic information of the subjects, whether they were born with jaundice, whether they have family member with autism, and ten behavioral features (AQ-10) recorded from the ASD screening. We only use the AQ-10 features, ages, and genders, of the subjects to determine their ASD condition.

5. Cervical Cancer (Fernandes, Cardoso, and Fernandes, 2017). The dataset was obtained from ‘Hospital Universitario de Caracas’ in Caracas, Venezuela. The dataset contains demographic information, habits, and historic medical records of 858 patients. The goal is to determine if a patient has cancer or not.

All datasets are split into 70% training, 15% validation, and 15% testing. We report the model performance in the testing dataset. The model settings are as follows.

- In all experiments, we use a DSE-KNN architecture of 2 embedding layers and 3 similarity layers. Each embedding layer has $4 \times d$ neurons and each similarity layers has $8 \times d$ neurons, with $d$ being the dimensionality of the
data. We start training with a learning rate of 0.1, and decrease by a factor of 10 when training cost fluctuates (due to the learning rate being too high for current training epoch). Model training is stopped when the change in training cost is negligible.

• SVM models use RBF kernels of which hyper-parameters are optimized via grid-search on the validation dataset.

• Gradient Boosting model and Random Forest model both have 200 estimators. We fit each model ten times and select the one with the highest validation performance to apply on the testing sets.

• Multiple DNN models with one to six layers, each has $5 \times d$ neurons, are trained with the training data. We then select the architectures with the highest validation accuracy to apply to the testing data.

Table 7.1 shows the test accuracy rates for all the models. As can be seen, DSE-KNN achieves the best results in all experiments (including two cases with equal performances from other models).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DSE-KNN</th>
<th>SVM</th>
<th>GB</th>
<th>RF</th>
<th>DNN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pima Diabetes</td>
<td>73.96</td>
<td>69.79</td>
<td>67.18</td>
<td>69.79</td>
<td>71.86</td>
</tr>
<tr>
<td>Diabetic R. Debrecen</td>
<td>81.60</td>
<td>77.78</td>
<td>70.49</td>
<td>71.68</td>
<td>76.39</td>
</tr>
<tr>
<td>Breast Cancer</td>
<td>97.14</td>
<td>95.24</td>
<td>94.29</td>
<td>95.24</td>
<td>97.14</td>
</tr>
<tr>
<td>Autism Screening</td>
<td>98.11</td>
<td>92.45</td>
<td>97.17</td>
<td>95.28</td>
<td>86.79</td>
</tr>
<tr>
<td>Cervical Cancer</td>
<td>95.35</td>
<td>95.35</td>
<td>93.80</td>
<td>94.57</td>
<td>95.35</td>
</tr>
</tbody>
</table>
7.4 Conclusion

In this chapter, we propose a new supervised learning method that is called DSE-KNN, which utilizes DEK architecture to automatically learn pairwise similarity function from data that is optimized for SE-KNN decision making. A unique feature of DSE-KNN is that the similarity function of data is automatically learned and this learning is directly driven by the goal of maximizing KNN decision accuracy. The learned similarity function will implicitly map data to a high-dimensional feature space where the quality of neighborhoods is optimized for KNN decision making. Experimental results show that DSE-KNN outperforms or achieves equal best performances to other common machine learning methods on classifying different types of disease datasets.
Chapter 8

Recurrent Embedding Kernel for Time Series Classification

8.1 Introduction

Stock price predictions are interesting, potentially profitable, yet very challenging tasks that have attracted a lot of researchers over the decades. Analytical methods, including statistical time series models, traditional machine learning approaches, and the more recent deep recurrent neural networks have been used in modeling and predicting stock price directions (Adhikari and Agrawal, 2014) (Liew and Mayster, 2017) (Qian, 2017) (Fischer and Krauss, 2018) (Fičura, 2017)(Nelson, Pereira, and Oliveira, 2017) (Shen et al., 2018). The study in (Qian, 2017) showed that machine learning approaches surpasses traditional time series models in precision on financial series prediction. The results in (Fischer and
Krauss, 2018), (Nelson, Pereira, and Oliveira, 2017), and (Shen et al., 2018) further demonstrated that recurrent architectures such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU), as a state-of-the-art technique for sequence learning, outperformed those memory-free machine learning approaches. In this chapter, we analyze the limitation of those recurrent neural networks and propose a novel deep architecture called Recurrent Embedding Kernel that is able to outperform LSTM and other recurrent neural networks on predicting stock daily price direction.

Statistics models for time series include Box-Jenkins Autoregressive Integrated Moving Average (ARIMA), State Space Model (SSM), Autoregressive Conditional Heteroskedasticity (ARCH) and Generalize ARCH (GARCH) (Hamilton, 1994). In general, these models require making certain assumption on the data (e.g., ARIMA models assume the series to be stationary), and may fail if their assumptions are not met. Moreover, statistical models cannot handle nonlinearities in the data without user’s specifications (e.g., selecting the differencing term in ARIMA). Both facts help statistical methods achieve good interpretability, but also limit their predicting power. Traditional machine learning approaches such as Support Vector Machines (SVM) (Cortes and Vapnik, 1995), Random Forest (RF) (Liaw and Wiener, 2002), and Multi-layer Perceptron (MLP) (Kruse et al., 2013) come with fewer to no data assumptions and provide more power in modeling nonlinearities of data. Therefore, they often outperform statistical models in prediction accuracy. However, those machine learning models still require certain important controls from users. For examples, SVM needs the selection of a kernel function and the tuning of hyper-parameters; SVM, RF,
and MLP cannot automatically determine how much historical data is needed for decision making.

Recurrent neural architectures were particularly designed for learning from sequential data with variable lengths. Common types of recurrent architectures include vanilla Recurrent Neural Network (RNN) (Funahashi and Nakamura, 1993), Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997), and Gated Recurrent Unit (GRU) (Chung et al., 2014). This type of recurrent architectures uses hidden states and/or gates to learn to encode a historical sequence into a memory vector which is then used for decision making at the current time point. However, encoding the entire history into a vector may unavoidably cause information loss regardless of memory learning and updating mechanisms. For those tasks, where a decision needs to be made on the current time point and similar historical time points are of great references to the decision making, it may be more beneficial to keep the entire history for decision making instead of vectorizing it.

In this chapter, we propose the Recurrent Embedding Kernel (REK) to address the issue of limited memory capacity of recurrent neural networks and apply it to predict stock daily price direction. REK is a recurrent of the Deep Similarity-Enhanced K Nearest Neighbors (DSE-KNN) discussed in Chapter 7. Unlike RNN, LSTM, and GRU, REK makes decision based on the entire history to minimize information loss. Like DSE-KNN, REK consists of three components, a Recurrent Embedding Network (REN), and a Kernel Network (KN), and the decision making model Similarity-Enhanced K Nearest Neighbors (SE-KNN). The REN component learns to vectorize the memory state of each time
point in the given sequence; whereas the KN component learns a kernel function that implicitly maps vectors learned by REN to a high-dimensional feature space, which is optimized for decision making through SE-KNN. Both the REN and the KN components are trained simultaneously with the same learning objective of minimizing the KNN loss. By minimizing KNN loss, REK learns to map the data sequence to a feature space where SE-KNN decision accuracy is maximized. Experimental results on multiple stock ETFs with different long-term trends show that REK outperforms vanilla RNN, LSTM, and GRU, on predicting daily price direction.

8.2 Recurrent Neural Networks for Stock Prediction

In this chapter, we work on the task of predicting stock price direction for the next trading day. In other words, given the price sequence of a stock ticker up to the current day, we predict if the price would be up or down for the next trading day. Obviously, the prediction not only depends on the price of the current day but also on certain critical information from historical price movements. Recurrent neural networks (including vanilla RNN, LSTM, and GRU) were designed to keep certain memories of historical information that will be incorporated with the current input for decision making. Therefore, we will first empirically evaluate the performances of these recurrent neural networks on this task.
8.2. Recurrent Neural Networks for Stock Prediction

8.2.1 Vanilla RNN

In vanilla RNN’s, the memory state of the current time point is computed from both the current input and its previous memory state. More formally, given a sequence \( X = \{X^{(1)}, X^{(2)}, \ldots, X^{(t)}, \ldots, X^{(T)}\} \), the hidden state \( U^{(t)} \) at time \( t \) that is outputted by the network can be expressed as

\[
U^{(t)} = g(W \cdot X^{(t)} + R \cdot U^{(t-1)} + b) \tag{8.1}
\]

where \( X^{(t)} \) is the input state of the sequence at the time point \( t \); \( W \) and \( R \) are weight matrices of the network; \( b \) is the bias vector of the network; and \( g(\cdot) \) is a selected activation function. RNN can be deepened by either stacking multiple RNN layers or increasing the number of layers to compute \( U^{(t)} \) from \( X^{(t)} \) and \( U^{(t-1)} \). Specifically for the task of predicting daily stock price direction, we have one binary output \( y^{(t)} \) for each time point \( t \) (i.e. each day in the sequence) which is computed from the memory state \( U^{(t)} \) of day \( t \). The computational flow of RNN is shown in Fig. 8.1. Since its memory state is updated with the current input at every time point, vanilla RNN is typically unable to keep long-term memory.

8.2.2 LSTM

LSTM is an improved version of RNN with the design goal of learning to capture both long-term and short-term memories. A LSTM block, shown in Fig. 8.2, uses gates to control how much its long-term memory would be updated at each
time point. The outputted short-term memory is then computed from the current input, the current long-term memory, and the previous short-term memory. More formally, an LSTM block can be described by the following formula:

\[
Z(t) = g(W_Z \cdot X(t) + R_Z \cdot U(t-1) + b_Z) \tag{8.2}
\]

\[
i(t) = \sigma(W_i \cdot X(t) + R_i \cdot U(t-1) + p_i \times C(t-1) + b_i) \tag{8.3}
\]

\[
f(t) = \sigma(W_f \cdot X(t) + R_f \cdot U(t-1) + p_f \times C(t-1) + b_f) \tag{8.4}
\]

\[
C(t) = i(t) \times Z(t) + f(t) \times C(t-1) \tag{8.5}
\]

\[
o(t) = \sigma(W_o \cdot X(t) + R_o \cdot U(t-1) + p_o \times C(t-1) + b_o) \tag{8.6}
\]

\[
U(t) = o(t) \times h(C(t)) \tag{8.7}
\]

where \(W_*\) and \(R_*\) are weight matrices; \(b_*\) are bias vectors; \(p_*\) are peepholes; \(X(t), U(t),\) and \(C(t)\) are the LSTM’s input, output, and cell state (i.e. long-term
memory) at time point $t$; $Z^{(t)}$ is the proposed update to the cell state; $i^{(t)}$, $f^{(t)}$, and $o^{(t)}$ are the output of the input gate, forget gate, and output gate, respectively; $g(\cdot)$ is the input activation, $\sigma(\cdot)$ is the sigmoid function, and $h(\cdot)$ is the output activation. Similarly to vanilla RNN, the LSTM architecture utilized in this chapter has one binary output for each day in the sequence. The simplified computational flow of a LSTM block is shown in Fig. 8.2. Note that the dashed arrows represent the computation of the elements being not direct but instead controlled by gate functions.

![Figure 8.2: The Computational Flow of LSTM for Predicting Daily Stock Price Direction](image)

Compared with vanilla RNN, LSTM introduces a mechanism to learn to capture task-relevant long-term memory. At each time point, the captured long-term memory is expressed as a vector. However, it is questionable, for a given task such as the discussed stock prediction, whether this vector is able to encode enough information that is critical to the task.
8.2.3 GRU

The architecture of an LSTM block is rather complex, which may cause training of the LSTM-based model difficult and time consuming. GRU can be viewed as an alternative to LSTM that can learn to capture task-relevant long-term memories with a simplified architecture. A GRU block contains only two gates. The memory states computed by GRU can be mathematically described using the following formula:

\[
\begin{align*}
U^{(t)} &= (1 - z^{(t)}) \times U^{(t-1)} + z^{(t)} \times \tilde{U}^{(t)} \\
\tilde{U}^{(t)} &= g(W_U \cdot X^{(t)} + R_U \cdot (r^{(t)} \times U^{(t-1)}) + b_U) \\
z^{(t)} &= \sigma(W_z \cdot X^{(t)} + R_z \cdot U^{(t-1)} + b_z) \\
r^{(t)} &= \sigma(W_r \cdot X^{(t)} + R_r \cdot U^{(t-1)} + b_r)
\end{align*}
\]  

(8.8)

(8.9)

(8.10)

(8.11)

where all notations are similar to LSTM, except for \(z^{(t)}\) and \(r^{(t)}\), which are the outputs of the update gate and reset gate, respectively. The computational flow of the GRU architecture utilized in this chapter is similar to that of RNN (refer to Figure 8.1), however with the computations from \(X^{(t-1)}\) and \(U^{(t-1)}\) to \(U^{(t)}\) controlled by gates.

8.2.4 Empirical Results

We evaluated the three recurrent neural networks on the task of predicting stock daily price direction. More specifically, we trained the models of these three architectures on daily data of six ETFs (SPY, DIA, IYR, GLD, VDE, and GDX) in the period of year 2008 to 2016 (data of year 2016 is used as the validation set...
for determining early stopping of the training), and tested the trained models on the daily data of year 2017. The test results show the three models have comparable performances (refer to Table 8.1 in Section 8.4). Surprisingly, although LSTM and GRU have mechanisms to learn to capture task-relevant long-term memory, they are not achieving consistently "better" performances than vanilla RNN. This raises the question that whether the ways that LSTM and GRU use to encode long term memory to vectors are sufficient to capture task critical information. Obviously, no matter how LSTM or GRU learn to encode historical data into memory vectors, this type of encoding leads to information loss.

This experimental result suggests that, instead of encoding the entire history into one or multiple vectors at the current time point, the entire history may be kept and searched for information that is critical for decision making. Keeping entire history by itself is not a complex task given the computing power of this age, but the challenge lies in how the model effectively identifies information that are critical to the task from the history. A simple strategy to retrieve information from the historical data is to use K Nearest Neighbors (KNN) method. KNN can be done based on different specifications of data representations and data relationships. Obviously, different specifications will lead to different KNN decision accuracy. In this paper, we will describe a novel deep architecture that is called Recurrent Embedding Kernel (REK), which automatically learns both optimal representation of each time point and optimal relationship among time points directly towards the goal of maximizing KNN decision accuracy. The details of REK will be provided in the following section.
Chapter 8. Recurrent Embedding Kernel for Time Series Classification

8.3 Recurrent Embedding Kernel

In brief, REK is a deep architecture that consists of two components: a Recurrent Embedding Network (REN) and a Kernel Network (KN). We first describe how a REK makes decision after it is trained. The REN component first vectorizes the state of every time point in the given sequence. In order to make decision at the current time point \( t \), the KN component computes the similarity between the vector of the current time point and each of the vectors at historical time points. Then, SE-KNN is used to make the decision based on the similarities output by the KN component. The decision making process of REK is illustrated in Figure 8.3.

Like DSE-KNN, the training of REK is guided towards the goal of maximizing SE-KNN decision making accuracy. This is done using the KNN Loss function as the learning objective, and updating layers in REK using Gradient Descent.

8.3.1 Similarity-Enhanced KNN and KNN Loss

The SE-KNN component and KNN Loss are as described in Chapter 7. In short, let the similarity of \( X^{(i)} \) and \( X^{(j)} \) in a feature space \( S \) be represented through a kernel function \( K(\cdot, \cdot) \). Then in \( S \), \( X^{(i)} \) and \( X^{(j)} \) are similar if \( K(X^{(i)}, X^{(j)}) \) is high; \( K(X^{(i)}, X^{(j)}) \) are dissimilar if \( K(X^{(i)}, X^{(j)}) \) is low. For each instance \( X^{(i)} \), the probability of its label \( y^{(i)} \) belonging to class \( c_l \) among the set of all classes \( \{c_1, c_2, ..., c_n\} \) is computed from its neighborhood of \( k \) most similar instances in
8.3. Recurrent Embedding Kernel

**Figure 8.3:** The Decision Making Process of Recurrent Embedding Kernel
Chapter 8. Recurrent Embedding Kernel for Time Series Classification

\[ P(y^{(i)} = c_l) = \frac{\sum_{j \in KNN(i)} K(X^{(i)}, X^{(j)})}{\sum_{j \in KNN(i)} K(X^{(i)}, X^{(j)})} \]  

(8.12)

The KNN loss is then designed to quantify the decision making accuracy of SE-KNN in a loss function. To compute KNN loss \( L^{(i)} \) at \( X^{(i)} \), first we label the pair \( X^{(i)} \) and each \( X^{(j)} \) as \( Y^{(i,j)} = 1 \) if \( y^{(i)} = y^{(j)} \), and \( Y^{(i,j)} = 0 \) otherwise. We further define the training neighborhood \( tKNN^{(i)} \) of \( X^{(i)} \) as the neighborhood of \( k \) most similar instances of the same label as \( X^{(i)} \). The KNN Loss in \( KNN^{(i)} \) of \( X^{(i)} \) in the feature space \( S \) can then be mathematically expressed as the binary cross entropy function of \( Y^{(i,j)} \) and \( K(X^{(i)}, X^{(j)}) \):

\[
L^{(i)} = - \sum_{j \in tKNN^{(i)}} (Y^{(i,j)} \log(K(X^{(i)}, X^{(j)})) + (1 - Y^{(i,j)}) \log(1 - K(X^{(i)}, X^{(j)}))) 
\]

(8.13)

The KNN loss over the whole dataset is then the sum of losses from every instance

\[
L = \sum_{i \in data} L^{(i)} 
\]

(8.14)

8.3.2 Kernel Network and Recurrent Embedding Network

The top component of REK is a kernel network (KN) with architecture as described in Chapter 5. KN learns to map data to an implicit dimensional feature space where decision is made. The output of KN is the similarity value of a pair of instances, represented through the probability of the pair having the same labels.
8.3. Recurrent Embedding Kernel

KN takes input as a vector transformed from two embedding vectors output by the bottom component Recurrent Embedding Network (REN). The purpose of REN is to learn optimal embedding of the state of each time point. Then, the learned embedding are used as input to the KN that is described above. To learn embedding for time points in a time series, we can use vanilla RNN, LSTM, or GRU. This work uses vanilla RNN for the following reasons. First, through experimental studies that are described in Section 8.2.4, vanilla RNN, LSTM, and GRU produced comparable results on predicting stock daily price direction. Second, vanilla RNN is simpler in architecture and thus easier to train.

Both REN and KN are trained as one integrated network using KNN loss. That means both embedding learning and kernel learning target the same objective of implicitly mapping data to a space that is optimized for similarity-enhanced KNN.

By stacking KN on top of REN, we have the complete REK. Fig. 8.4 illustrates the computational flow of REK in calculating the kernel value $K(X^{(t)}, X^{(v)})$ for two time points $t$ and $v$. In details, let $t < v$ be two time points in the sequence, $\hat{E}(\cdot)$ be the REN component, and $\hat{K}(\cdot)$ be the kernel network component, then we have

$$U^{(t)} = \hat{E}(X^{(t)}, U^{(t-1)})$$  \hspace{1cm} (8.15)

$$U^{(v)} = \hat{E}(X^{(v)}, U^{(v-1)})$$  \hspace{1cm} (8.16)

$$U^{(t,v)} = \{|U^{(t)} - U^{(v)}| \cup (U^{(t)} \times U^{(v)})\}$$  \hspace{1cm} (8.17)

$$K(X^{(t)}, X^{(v)}) = \hat{K}(U^{(t,v)})$$  \hspace{1cm} (8.18)
The training process of REK on a training sequence \( X = \{ X_0, X_1, ... X^{(t)}, ... \} \) can be described as follows.

1. The whole sequence is input into REN that outputs the embedding for each time point ("Generating Embedding" step in Fig. 8.5)

2. Every pair of embedding vectors are fed into the KN that outputs its pairwise similarity ("Computing Pairwise Similarity" step in Fig. 8.5)

3. The training neighborhood \( tKNN^{(t)} \) is formed for each \( X^{(t)} \) ("Forming \( tKNN^{(t)} \)" step in Fig. 8.5)

4. The KNN loss at \( X^{(t)} \) is computed using equation (8.13) ("KNN Loss for \( X^{(t)} \)" step in Fig. 8.5)
5. The aggregated KNN loss is calculated using equation (8.14) ("Aggregating KNN Loss" step in Fig. 8.5)

6. Use Gradient Descent to update the weight matrices and bias vectors throughout the REK network in order to minimize the aggregated KNN loss (illustrated by the backward arrows in Fig. 8.5).

To prevent overfitting, we remove time points which have the same labels with all their $k$ nearest neighbors. Moreover, we observe that selecting training
neighborhoods for all instances in every epoch results in training instabilities and slower convergence speeds. Therefore, we fix the $tKNN^{(i)}$ of all $i$’s for a certain number of training epochs (e.g., 50 or 100 epochs) before we re-calculate $tKNN^{(i)}$ based on the new weights of the REK.

## 8.4 Experiments

We implement all experiments in Python 2.7.15rc1. All deep models are implemented using Theano (Bergstra et al., 2010); Visualization of ETFs are done using the Matplotlib package (Hunter, 2007). We test our model using six ETFs:

1. SPDR S&P 500 ETF Trust (SPY)
2. SPDR Dow Jones Industrial Average ETF (DIA)
3. iShares US real Estate ETF (IYR)
4. SPDR Gold Shares (GLD)
5. Vanguard Energy ETF (VDE)
6. VanEck Vectors Gold Miners ETF (GDX)

All stock data are obtained from Yahoo! Finance from the beginning of 2008 to the end of 2017. We select the six ETFs in order to cover different types of long-term trends. As shown in Fig. 8.6, SPY, DIA, and IYR have an overall upward trend; GLD presents an uptrend followed by a downtrend; VDE develops a long-term consolidation; and GDX demonstrate a downward trend for the most part of the series.
Our task is to predict if the adjusted close price of the next day is up or down. For this purpose, we label each day in the training sequence as "Up" if the adjusted close of its next day is higher than its current adjusted close price; otherwise, we label it as "Down". We use 18 input features to describe each day, which include open, close, highest, lowest, adjusted close, and volume, of the current day, current week, and current month.
Given that the stock price can change dramatically over a long term period, in order to avoid the sharp differences in the input values of any two given days, we use the differences of current day’s values and previous day’s values as the input values for each feature.

For all sequences, data from 2008 to 2016 is used as training sets, and data in 2017 is used as testing sets. The training set is further split into training (2008 to 2015) and validation (2016) for determining early stop of the training. We train REK’s using the proposed KNN loss. We choose $k = 25$ in all experiments. The REK that is used in all experiments includes a two-layer REN (with 36 hidden neurons each layer), and a three-layer KN (one hidden layer of 36 neurons and one output neuron).

We compare REK with LSTM, GRU, and vanilla RNN models using the same training, validating, and testing data. LSTM and GRU models have 36 hidden neurons. Vanilla RNN models have two hidden layers of 36 neurons. All referenced models use Softmax for decision making. We also tested vanilla RNNs with three or more hidden layers but they fail to converge.

All models are trained with decaying learning rates starting from 0.1. We then decrease the learning rates by a factor of 10 when training cost begins to heavily fluctuate among adjacent epochs. Model training is controlled using early stopping with validation accuracy. The last model with highest validation accuracy is then fitted on the testing data to obtain the testing accuracy. All models use ReLU activation, except for LSTM that uses tanh, because ReLU LSTMs explode rapidly in our experiments.

Table 8.1 shows the testing performance of all models on each ETF. We also
include the percentage of actual Up days in the testing data in the second column of Table 8.1. As can be seen, the percentages of actual up days for the six ETFs are falling in the range of 46% to 62%, which indicates that guessing one single class label for all testing instance cannot produce meaningful results. As shown in Table 8.1, REK outperforms other models on six ETFs. REK consistently obtains accuracy rates over 70% in five ETFs, and still gives reasonable performance (about 67.5%) on GDX whereas other models perform significantly worse on this ETF.

<table>
<thead>
<tr>
<th>Data</th>
<th>% Actual Up Days</th>
<th>REK</th>
<th>RNN</th>
<th>GRU</th>
<th>LSTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPY</td>
<td>57.32</td>
<td>71.43</td>
<td>67.55</td>
<td>69.11</td>
<td>69.11</td>
</tr>
<tr>
<td>DIA</td>
<td>61.45</td>
<td>71.89</td>
<td>67.55</td>
<td>70.27</td>
<td>68.07</td>
</tr>
<tr>
<td>IYR</td>
<td>53.01</td>
<td>70.06</td>
<td>67.47</td>
<td>65.86</td>
<td>69.08</td>
</tr>
<tr>
<td>GLD</td>
<td>56.63</td>
<td>70.28</td>
<td>67.46</td>
<td>66.36</td>
<td>70.04</td>
</tr>
<tr>
<td>VDE</td>
<td>47.79</td>
<td>71.89</td>
<td>64.23</td>
<td>69.88</td>
<td>70.28</td>
</tr>
<tr>
<td>GDX</td>
<td>50.60</td>
<td>67.47</td>
<td>61.45</td>
<td>55.02</td>
<td>60.24</td>
</tr>
</tbody>
</table>

8.5 Conclusion

In this chapter, we apply the Deep Similarity-Enhanced K Nearest Neighbors framework that is discussed in Chapter 7 into time series data. This version of DSE-KNN is called Recurrent Embedding Kernel. Going beyond the limitation of memory capacity that the state-of-the-art recurrent architectures can utilize, this new model learns to vectorize each historical time point and learns to make optimized decision based on the entire history. The learning objective of REK is
to implicitly map time points in a given sequence to a high level feature space that is optimal towards decision making. REK simultaneously learns both optimal representations of time points and optimal pairwise similarity between time points towards this learning objective. More specifically, this learning objective is to minimizing the KNN loss, which in turns maximizes the decision accuracy using a similarity-enhanced KNN strategy.

Experimental results on multiple stock ETFs with different long-term trends show that REK outperforms state-of-the-art recurrent architectures on predicting daily price direction. While this paper focuses on the application of predicting stock price direction, REK can also be applied to other time series tasks.
Chapter 9

Conclusion

This dissertation addresses the advantages and disadvantages in kernel methods and deep learning, the two major branches of supervised machine learning, through a series of algorithms.

The proposed algorithms center around the Deep Embedding Kernel (DEK) architecture – a deep network realization of a kernel function. DEK consisting of two components: an embedding network and a kernel network. The embedding network takes in individual data instances and output their embedding vectors. The kernel network then takes in pair of embedding vectors and output a single scalar of the pair of instances. The embedding network and the kernel network are transparently connected and can be trained end-to-end during a single process of gradient descent. DEK is designed to be symmetric and positive definite so that it can represent a true kernel function. In the chapter proposing DEK, experimental studies show that DEK outperforms and state-of-the-art machine learning and deep learning algorithms in the tasks of general classification and dimension reduction.
Through DEK, data can be mapped to an implicit feature space with characteristics that are specified by a learning goal represented by a loss function. The original DEK focuses on general classification. Consequently, the learning goal is to optimize the pairwise similarity among data instances in the feature space and is represented through the Binary Cross-Entropy loss function. More recently, my other works, Recurrent Embedding Kernel (REK) and Deep Similarity-Enhanced K Nearest Neighbors (DSE-KNN), utilize a mapping that is optimized for neighborhoods of data instances in feature spaces. To do this, we propose a new loss function, KNN Loss, to guide the training of deep architectures towards optimizing the neighborhood in feature space. Experimental studies show that REK outperforms Recurrent Neural Networks (RNN) (including vanilla RNN, Long Short-Term Memory, and Gated Recurrent Unit) in predicting daily stock price movements; and DSE-KNN outperforms traditional machine learning methods in classification of disease data.

Another extension of DEK is in solving problems in big data. In such cases, although new models can always be trained, the users may not always have access to the necessary computational resources. Moreover, numerous applications have already had pretrained architectures made public that can be utilized in the new models for the same tasks but in new data. DEK is integrated with the Dual Deep Learning framework for this purpose. The overall framework consists of two phases with two different deep networks: data representation learning, and data relationship learning. The first phase aims to generate a knowledge background for the second phase, thus, any deep networks (including public pretrained networks) suitable for the data type can be used. The
second phase uses DEK to learn the data relationship optimized for the given task. We show that the DEK/DDL framework outperforms Google Facenet in facial recognition in the Indian Movie Face Database set.
Bibliography


Fičura, Milan (2017). “Forecasting Foreign Exchange Rate Movements with k-Nearest-Neighbour, Ridge Regression and Feed-Forward Neural Networks”. In:


Bibliography


Kandola, Jaz, John Shawe-Taylor, and Nello Cristianini (2002). “Optimizing kernel alignment over combinations of kernel”. In:


Liew, Jim Kyung-Soo and Boris Mayster (2017). “Forecasting ETFs with Machine Learning Algorithms”. In:


Ng, Andrew (2012). “Clustering with the k-means algorithm”. In: Machine Learning.


Toker, Gülen and O Kirmemis (2013). “Text categorization using k nearest neighbor classification”. In: *Survey Paper, Middle East Technical University*.


Bibliography


Xu, Shizhen and Yundong Wu (2008). “An algorithm for remote sensing image classification based on artificial immune B-cell network”. In: The international archives of the photogrammetry, remote sensing and spatial information sciences 37, pp. 107–112.


