Studies in Computational Intelligence 866

Witold Pedrycz
Shyi-Ming Chen *Editors*

Deep Learning: Concepts and Architectures

Springer

Studies in Computational Intelligence

Volume 866

Series Editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland

The series “Studies in Computational Intelligence” (SCI) publishes new developments and advances in the various areas of computational intelligence—quickly and with a high quality. The intent is to cover the theory, applications, and design methods of computational intelligence, as embedded in the fields of engineering, computer science, physics and life sciences, as well as the methodologies behind them. The series contains monographs, lecture notes and edited volumes in computational intelligence spanning the areas of neural networks, connectionist systems, genetic algorithms, evolutionary computation, artificial intelligence, cellular automata, self-organizing systems, soft computing, fuzzy systems, and hybrid intelligent systems. Of particular value to both the contributors and the readership are the short publication timeframe and the world-wide distribution, which enable both wide and rapid dissemination of research output.

The books of this series are submitted to indexing to Web of Science, EI-Compendex, DBLP, SCOPUS, Google Scholar and Springerlink.

More information about this series at http://www.springer.com/series/7092

Witold Pedrycz · Shyi-Ming Chen
Editors

Deep Learning: Concepts and Architectures

Springer

Editors

Witold Pedrycz
Department of Electrical
and Computer Engineering
University of Alberta
Edmonton, AB, Canada

Shyi-Ming Chen
Department of Computer Science
and Information Engineering
National Taiwan University of Science
and Technology
Taipei, Taiwan

ISSN 1860-949X ISSN 1860-9503 (electronic)
Studies in Computational Intelligence
ISBN 978-3-030-31755-3 ISBN 978-3-030-31756-0 (eBook)
https://doi.org/10.1007/978-3-030-31756-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Deep learning delivers an interesting and conceptually as well as algorithmically state-of-the-art approach to Artificial Intelligence and Intelligent Systems, in general. This paradigm has been applied to numerous areas including machine translation, computer vision, and natural language processing. Deep learning, regarded as a subset of machine learning, utilizes a hierarchy level of artificial neural networks to carry out efficiently the process of machine learning.

This volume provides the reader with a comprehensive and up-to-date treatise in the area of deep learning. There are two focal and closely associated aspects here, namely concepts supported by the environment of deep learning and a plethora of its architectures. Those two faculties are strongly intertwined as well as linked with the application domain under discussion. The concepts of deep learning revolve around the structural and automatic (to a significant degree) mechanisms knowledge representation. A variety of multilayer architectures bring about the tangible and functionally meaningful pieces of knowledge. Their structural development becomes essential to successful practical solutions. The architectural developments that arise here support their detailed learning and refinements.

The chapters, authored by active researchers in the field, bring a collection of studies that reflect upon the current trends in design and analysis of deep learning topologies and ensuing applied areas of successful realizations including language modeling, graph representation, and forecasting.

We would like to take this opportunity and express our thanks to the Series Editor-in-Chief, Professor Janusz Kacprzyk, who has played an instrumental role in the realization of the volume by providing constant encouragement and support. We are indebted to the enthusiastic team at Springer; those are the professionals who, as usual, delivered advice and guidance as well as made the entire production efficient and completed in a timely manner.

Edmonton, Canada — Witold Pedrycz
Taipei, Taiwan — Shyi-Ming Chen

Contents

Deep Learning Architectures

Mohammad-Parsa Hosseini, Senbao Lu, Kavin Kamaraj, Alexander Slowikowski and Haygreev C. Venkatesh

Abstract Deep learning is one of the most widely used machine learning techniques which has achieved enormous success in applications such as anomaly detection, image detection, pattern recognition, and natural language processing. Deep learning architectures have revolutionized the analytical landscape for big data amidst wide-scale deployment of sensory networks and improved communication protocols. In this chapter, we will discuss multiple deep learning architectures and explain their underlying mathematical concepts. An up-to-date overview here presented concerns three main categories of neural networks, namely, Convolutional Neural Networks, Pretrained Unsupervised Networks, and Recurrent/Recursive Neural Networks. Applications of each of these architectures in selected areas such as pattern recognition and image detection are also discussed.

Keywords Deep learning · Architectures · CNN · Unsupervised networks · Recurrent networks · Recursive networks · LSTM · Autoencoders · GAN · Attention · DBN

M.-P. Hosseini (✉) · S. Lu · K. Kamaraj · A. Slowikowski · H. C. Venkatesh
Santa Clara University, Silicon Valley, CA, USA
e-mail: mhosseini@scu.edu; parsa@cac.rutgers.edu; mh973@cac.rutgers.edu

S. Lu
e-mail: slu1@scu.edu

K. Kamaraj
e-mail: kkamaraj@scu.edu

A. Slowikowski
e-mail: aslowikowski@scu.edu

H. C. Venkatesh
e-mail: hchincholivenkatesh@scu.edu

M.-P. Hosseini
AI Research, San Jose, CA, USA

In Collaboration with Electrical and Computer Engineering Department,
Rutgers University, New Brunswick, NJ, USA

Radiology Department, Henry Ford Health System, Detroit, MI, USA

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_1

1 Background

Machine learning is a branch of artificial intelligence (AI) that entails algorithms which enable computer systems to infer patterns from data. Machine learning has a broad scope of applications including bioinformatics, fraud detection, financial market analysis, image recognition, and natural language processing (NLP).

Traditional statistical machine learning techniques are limited in their ability to process natural data in raw form because the patterns and the inferences that must be made between them are complicated. They requires a lot of work from humans to extract proper features in order to improve their performance. Representation learning is then developed as a set of methods that allows a machine to automatically discover the representations (features) needed for detection or classification of data.

Deep learning is a class of representation machine learning methods with multiple levels of representation. It is composed of several simple but non-linear modules that each transforms the representation from previous levels (starting with the raw input) into a representation at a higher, slightly more abstract level. With the composition of enough such transformations, very complex features and inferences can be learned. In general, all of the deep learning methods can be classified into one of three different categories, which are Convolutional Neural Networks (CNNs), Pretrained Unsupervised Networks (PUNs), and Recurrent/Recursive Neural Networks (RNNs). We will discuss them in detail in the following sections, but first, we discuss how to train a deep learning model.

2 Training Procedure

In both deep learning and machine learning, predictive models utilize various underlying algorithms to infer mathematical relationships from training data. There are mainly three types of learning methods, namely: supervised learning, unsupervised learning, and semi-supervised learning. In the section below, we will discuss each method in greater detail.

2.1 Supervised Learning

In supervised learning, the model is fed a training dataset containing both observations (i.e., inputs) as well as their corresponding outcomes (i.e., outputs) [11]. The model then infers the mathematical mapping from inputs to outputs which it can use to classify future input test data points [7, 13]. Backpropagation, short for backward propagation of errors, is a popular supervised learning technique used in many artificial neural network architectures. In backpropagation, the weights of the neural network's constituent nodes are adaptively reconfigured to improve prediction accuracy.

2.2 Unsupervised Learning

In unsupervised learning, the model is fed with unclassified training data (i.e., only inputs). Then, the model categorizes test data points into different classes by finding commonalities between them. For example, let us assume we have an animal classification scenario in which the dataset contains unclassified pictures of a variety of animals. The model will sort through the dataset and extract different features from the images in order to assist with classification. Some of the extracted features could include the color of the animal and the size of the animal to name a few. Using these features, the data may be grouped into different clusters. For example, images containing a dog will ideally fall into the same cluster based on the semblance of the extracted features; the same idea applies to the other animals found in the dataset.

2.3 Semi-supervised Learning

As its name suggests, semi-supervised learning inherits properties from both supervised learning and unsupervised learning. A semi-supervised data set primarily contains unclassified training data points along with small amounts of classified data. Semi-supervised models feature two important advantages. One, they are substantially more accurate than unsupervised models with the addition of a few classified data points. Two, they are significantly less laborious and time-intensive compared to supervised learning. Semi-supervised learning may refer to either transductive learning or inductive learning.

The goal of transductive learning is to infer the correct labels for the given unlabeled data. Self-training is a commonly used transductive method for semi-supervised learning. In this method, a classifier is first trained with the sample of labeled data. Then the classifier is used to classify the unlabeled dataset. Normally the most assured unlabeled data, along with their predicted labels, are appended to the training set. The classifier is re-trained with the new data and the process is repeated. This process of re-training the classifier by itself is called self-teaching or bootstrapping. The model is then ready for classifying test data points. On the other hand, inductive learning is used to deduce the mapping between input and output. Inductive generative models are possibly the oldest semi-supervised learning method. It assumes a model, $p(x, y) = p(y)p(x|y)$ where $p(x|y)$ is a recognizable mixture distribution. With large amounts of unlabeled data, the mixture components can be recognized; then, we ideally require just one labeled example per component to fully determine the mixture distribution.

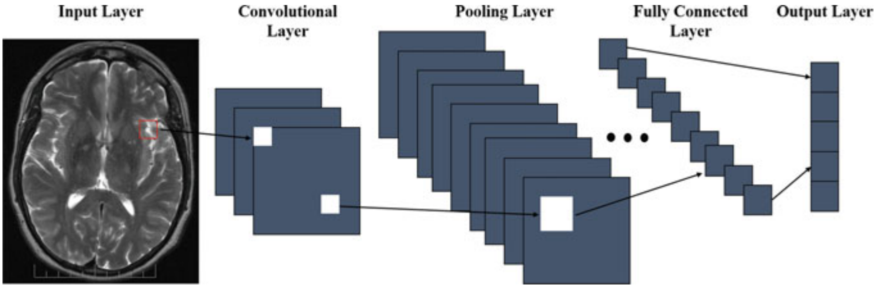


Fig. 1 A condensed representation of Convolutional Neural Networks (CNN). It is a type of feed-forward artificial neural network based on 3D neuronal arrangements, local connectivity between neurons of adjacent layers, and shared weight vectors

3 Deep Learning Categories

In this section below, we will discuss multiple deep learning architectures and explain their underlying algorithms. An up-to-date overview will be presented for each of the three main categories of neural networks, namely, Convolutional Neural Networks, Pretrained Unspervised Networks, and Recurrent/Recursive Neural Networks.

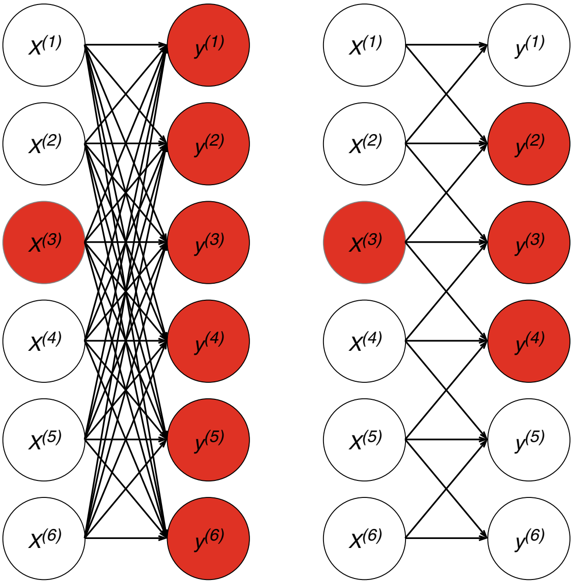
3.1 Convolutional Neural Networks (CNNs)

CNNs are inspired by biological processes and are designed to mimic the neural connectivity found in the brain's visual cortex. They requires considerably less data pre-processing compared to traditional image classification algorithms which require hand-engineered pre-processing filters [6]. CNNs have a large range of applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing (NLP).

3.1.1 CNN Structure

CNNs differ from conventional neural networks as they perform convolution instead of standard matrix multiplication in at least one of their layers (Fig. 1). They are famous for two distinct attributes: sparse interactions and parameter sharing. Sparse interactions or connectivity is achieved by making the model's kernel smaller than the size of the input. For example, in an image classification application, there may be millions of pixels representing a high resolution image. In this case, the kernel will be configured in a manner such that it only captures important features such as contrast and edges which are more indicative of the objects in the image. With fewer pixels of the image in consideration, there is a reduction in parameters to

Fig. 2 A network on the left without sparse and a network on the right using sparse with a kernel of 3



represent the image. This results in the reduction of memory utilization as well as computational overhead. On the other hand, traditional neural networks are less efficient and generate parameters for the interaction between each input unit and output unit [14]. Parameter sharing, also referred to as tied weights, involves tying the value of one input unit's weight to another input unit's weight. In the case of the image classification scenario, parameter sharing would ensure that there is only one set of parameters learned for each location of the image. This differs from traditional neural networks in which weights are not tied and separate sets of parameters are learned at each location.

Each layer of a convolutional network generally performs three steps. In the first step, the layer parallely performs multiple convolutions to generate a set of linear activations. After this, the second step—often referred to as the detector stage—entails running the linear activations through a non-linear activation functions. The objective of this is to ultimately infer a non-linear mapping to the output as needed. The detector stage is followed by a third stage comprised of a pooling function which makes further changes to the output of the layer. The purpose of a pooling function is to modify the output of a particular location in the net based on the statistical values of nearby outputs [9, 10, 18]. This emphasizes the convolutional aspect of CNNs in which neighborhood values have an impact on any given node. In the case of max pooling, the operation will modify an output value according to the max value of its rectangular neighborhood region. Other popular pooling functions consider the average value of the neighborhood region (Fig. 2).

Different weights are applied to different layers until the network is able to filter the data and achieve a result. This works by having the main convolutional layer pool and constructing feature maps based on different filters, or kernels. Each of these layers are fully connected and ultimately achieve an output. CNNs are mainly used for visual classification but can have many useful applications in text and language detection, object tracking, action recognition, and other classifications. Equation 1 below illustrates forward propagation implemented in a CNN, where ω is the n x n filter and ψ is the nonlinearity weight matrix, Eq. 2 represents the gradient component for each weight, and Eq. 3 represents the weights of the convolutional layer [6].

$$x_{ij}^{l} = \psi\left(\sum_{a=0}^{n-1}\sum_{b=0}^{n-1}\omega_{ab}\, y_{(i+a)(j+b)}^{l-1}\right) \tag{1}$$

$$\frac{\partial E}{\partial x_{ij}^{l}} = \frac{\partial E}{\partial y_{ij}^{l}}\frac{\partial y_{ij}^{l}}{\partial x_{ij}^{l}} = \frac{\partial E}{\partial y_{ij}^{l}}\frac{\partial}{\partial x_{ij}^{l}}(\psi(x_{ij}^{l})) \tag{2}$$

$$\omega_{ab} = J\left(\sum_{a=0}^{n-1}\sum_{b=0}^{n-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}}\frac{\partial x_{(i-a)(j-b)}^{l}}{\partial y_{ij}^{l-1}}\right) \tag{3}$$

$$J = \left(\sum_{a=0}^{n-1}\sum_{b=0}^{n-1}\frac{\partial E}{\partial x_{(i-a)(j-b)}^{l}}\right)^{-1} \tag{4}$$

3.1.2 CNN Architectures and Applications

Since the initial development of CNN, multiple CNN architectures have been created. Some notable examples include: LeNet-5, AlexNet, ResNet, and GoogLeNet. Each of these networks still employ the same structure of convolutional layers and feature extraction, but may vary in the number of layers they have, feature mapping, and efficiency.

LeNet [19] is the first successful application of convolutional networks and was developed by Yann LeCun in the 1990s. Of the CNNs, the best known is the LeNet architecture which was used to read zip codes, digits, etc. The latest work is called LeNet-5 which is a 5-layer CNN that reaches 99.2% accuracy on isolated character recognition [20].

The first work that popularized convolutional networks in computer vision was the AlexNet, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton for the University of Toronto [18]. The main difference is that AlexNet has multiple convolutional layers followed by a POOL layer. It contains eight main layers, where the first five layers are convolutional layers and the last three layers are fully connected layers. Following the convolutional layers, there are additional max-pooling layers that use the non-saturating Rectified Linear Unit (ReLU) activation function instead of either the hyperbolic tangent or sigmoid activation functions. Using ReLU increases the speed of AlexNet by a factor of six while maintaining similar accuracy.

Residual Neural Network (ResNet) was proposed in [4] by Microsoft Research. The layers are reformulated while learning residual functions instead of unreferenced functions. As a result these residual networks are easier to optimize and gain considerable accuracy from increasing network depth.

GoogLeNet is a convolutional network designed by Szegedy et al. from Google [29]. It is a much deeper CNN compared to AlexNet; GoogLeNet contains 22 layers compared to AlexNet's 8 layers. GoogLeNet's main contribution is the development of an inception layer that reduces the number of parameters in the network. Additionally, GoogLeNet uses the average pooling method at the top of the convolutional layers which further eliminates parameters that contribute minimal performance gains. It is worth noting that GoogLeNet contains 4 million parameters compared to AlexNet's 60 million parameters. There are several followup versions to the GoogLeNet, the most recent of which is Inception-v4.

While CNNs are versatile in various problem spaces, they still have limitations in their architectures. CNNs are known to be prone to overfitting and getting stuck at local minima. Both issues lead to lower model performance and higher computational time. Therefore optimization algorithms can be taken into consideration to help compensate for the limitations of CNNs. An extended optimization approach for CNN is proposed by Hosseini et al. [6] based on principle component analysis (PCA), independent component analysis (ICA), and Differential Search Algorithm (DSA). The proposed method can improve CNN's efficiency and optimize its feature extraction—particularly when dealing with a large scale complex dataset.

3.1.3 Forward and Backward Propagation

When data is inputted into a neural network, it initially advances through the network through a series of layers in what is referred to as forward propagation. The network can have various numbers of layers within the network which represents the depth of the network, and also includes an input vector, x. Each layer, l, also has a width which represents the number of nodes. Within each layer, we apply the weight matrix to the activation function; the incoming data, a, is multiplied by a weight, w, and a bias, b, is added to the result. In Eq. 5, j is the output node and represents the *jth* node from the *lth* layer, and k represents the *kth* node from the previous layer, $l-1$, which serves as the input node [1]. The value w^l_{jk}, then, is the weight relationship that exists between the two nodes from both layers. When the weight matrix and bias are first setup, they are randomly initialized through a process known as parameter initialization, with a^0 being the layer containing the input data vector (Fig. 3).

$$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j \tag{5}$$

The weighted input for the *jth* node in the *lth* layer, z^l_j, is then fed into an activation function, f.

$$a^l_j = f(z^l_j) \tag{6}$$

The role of an activation function is to produce an output, or a mapping from an input real number to a real number within a specific range in order to determine whether or not the information within the node is useful, i.e., to activate the node or not. It is the activation function which creates the layered design to the neural network;

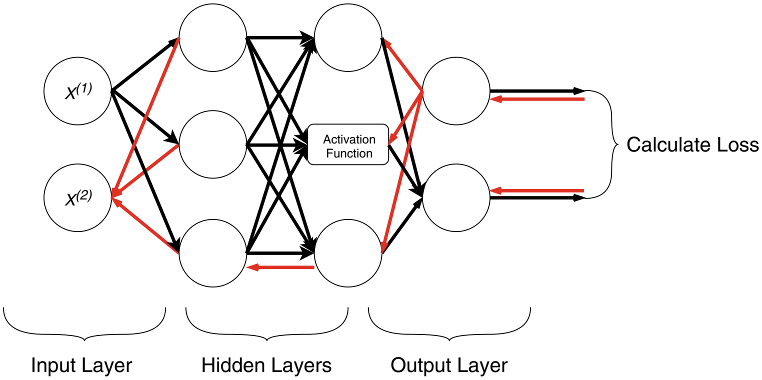


Fig. 3 A Neural Network using Back Propagation

an activation functions acts as a neural network layer by performing operations on the original input data and feeding it forward into the next neural network layer (activation function). The values produced by the activation function depend on the type of activation function which is used. There is not always a clear or best choice in deciding which activation function to use when designing the network; this process requires some trial and error to determine which activation function will produce the most optimum results. Some of the more common activation functions are listed below.

The sigmoid function, which is also known as the logistic function maps to real numbers with values ranging between 0 and 1.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

The hyperbolic tangent, $tanh$, activation function maps to real numbers with values ranging between -1 and 1.

$$f(x) = tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \tag{8}$$

The Rectified Linear Unit (ReLU) activation function maps to real numbers with values ranging from 0 to ∞.

$$f(x) = \{^{0 \ for \ x<0}_{x \ for \ x\geq 0} \tag{9}$$

The Softmax activation function maps to real numbers with values ranging from 0 to 1.

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{j} e^{x_j}} \tag{10}$$

The initialized parameter values which were arbitrarily set are adjusted later on as the network is trained. The data is said to be fed forward into the network and its final result produces an output vector, $\hat{y}$. The loss is then calculated using the output vector (see Eq. 12).

The forward propagation equation can be reduced to the simple vectorized equation

$$A^l = f(W^l A^{l-1} + b^l) \tag{11}$$

Backpropagation is one method of training a network by minimizing the loss as calculated by the cost function. The cost function represents how far off the network is from making accurate predictions based on the input. There are many different types of cost functions which may be used, one of which is the mean squared error (12) [1].

$$C = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{12}$$

Algorithm 1: Forward and Backward propagation in CNN

```
Input: M-dimensional data, $\mathbf{x} = [x_1, \cdots, x_N]^T$
begin
for $l := 1 \rightarrow \#HiddenLayers$ do
    for $i := 1 \rightarrow \#RowunitinLayerl$ do
        for $j := 1 \rightarrow \#ColumnunitinLayerl$ do
            Find the layer activations by, $y^l_{ij} = \varphi(x^l_{ij} + b^l_{ij})$
            Compute next layer inputs
        end
    end
end
Keep the final output as $y^l$
Calculate error at the output layer.
begin
for $l := \#HiddenLayers \rightarrow 1$ do
    Find error partial derivation by $\frac{\partial E}{\partial x^l_{ij}} = \frac{\partial E}{\partial y^l_{ij}}\frac{\partial y^l_{ij}}{\partial x^l_{ij}} = \frac{\partial E}{\partial y^l_{ij}}\frac{\partial}{\partial x^l_{ij}}(\psi(x^l_{ij}))$
    Find error at the previous layer.
end
Calculate the gradient of the error by $\partial\omega_{ab} = \sum_{i=0}^{N-n}\sum_{i=0}^{N-n}\frac{\partial E}{\partial x^l_{ij}}\frac{\partial x^l_{ij}}{\partial\omega_{ab}} = \sum_{i=0}^{N-n}\sum_{i=0}^{N-n}\frac{\partial E}{\partial x^l_{ij}}y^{l-1}_{(i+a)(j+b)}$
END
```

where n is the number of data points. The mean squared error can be understood as the average of the square of the difference between the desired, or expected output and the actual output which was obtained by the network [17].

In order to optimize the cost function by minimizing the loss, the gradient descent optimization algorithm is used and the errors are backpropagated up the chain toward the front of the network. Gradient descent adjusts each weight by taking the negative of the learning rate multiplied with the partial derivative of the cost function with respect to the weights [1].

$$\Delta w = -\eta \frac{\partial C}{\partial w} \tag{13}$$

With backpropagation, we are able to compute the gradients all at once by updating the weights using the chain rule [1]. Using the chain rule, the weights for each node are adjusted accordingly by multiplying the gradients together. Backpropagation has the added benefit that it greatly reduces the number of calculations required to compute the gradients compared to forward propagation. If forward propagation is used to update the weights, it suffers from the problem that for each weight, the cost function must first be calculated prior to calculating the partial derivative of the cost function with respect to the weight, resulting in a slow, intensive operation requiring the number of parameters, squared, iterations through the network to compute the gradients [26]. The network has to complete an entire forward iteration to calculate the gradient for each individual node. With backpropagation, a single forward propagation is performed to calculate the loss, and then a single backpropagation to update the weights with respect to the loss is all that is required to update the network. Backpropagation can also be understood from the pseudo-code in Algorithm 1.

3.2 Pretrained Unsupervised Networks

Data generation and feature extraction are very important applications in deep learning as we usually have limited training data. There are different techniques used to augment the initial dataset to provide a larger dataset from which to train the network. Using some of the most advanced deep learning architectures like Generative Adversarial Networks (GANs) and Autoencoders, we could generate synthetic data based off of the original dataset to improve model learning. Both architecture belongs to a family called Pretrained Unsupervised Network (PUN). PUN is a deep learning model that uses unsupervised learning to train each of the hidden layers in a neural network to achieve a more accurate fitting of the dataset. An unsupervised learning algorithm is used to train each layer one at a time, independently, while using the previously trained layer as the input. After the pre-training is done on each layer, a fine-tuning step is performed on the whole network using supervised learning. Types of PUNs include Autoencoders, Deep Belief Networks (DBN), and Generative Adversarial Networks (GAN) [25].

3.2.1 Autoencoders

Autoencoders use unsupervised learning to learn a representation for dimensionality reduction where the input is the same as the output (Fig. 4). The three parts of an autoencoder include the input, output, and the hidden layer. The data is compressed

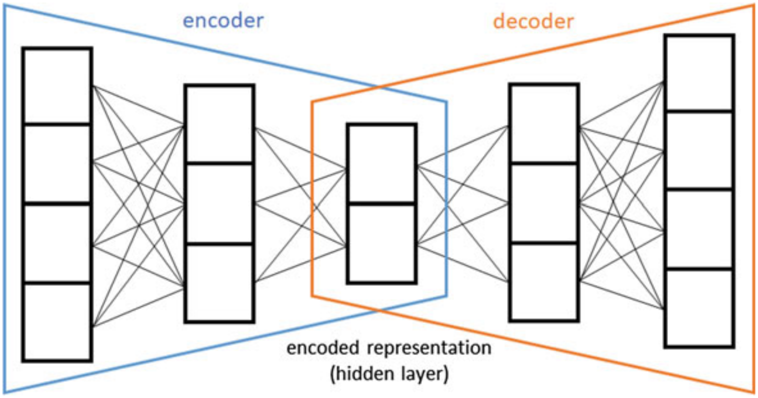


Fig. 4 A representation of an autoencoder. It uses unsupervised learning to learn a representation for dimensionality reduction where the input is the same as the output

into a smaller representation in the hidden layer then uncompressed to form an output that is similar to the input. This happens through two main steps of encoding and decoding of the autoencoder algorithm [15]. For example, the following is used to represent the mapping function between the input layer and hidden layer [24]:

$$y = f_{\Theta}(\hat{x}) = s(W\hat{x} + b) \tag{14}$$

where the input $\hat{x}$ is mapped to the hidden layer y, θ is the coding parameter, and W is the weighted matrix. Therefore the decoding function would be the following:

$$z = g_{\Theta'}(y) = x(W'y + b') \tag{15}$$

z would be the reconstruction of input x [24].

Autoencoders are a variant of feed-forward neural networks that have an extra bias for calculating the error of reconstructing the original input [9]. After training, autoencoders are used as a normal feed-forward neural network for activations. This is an unsupervised form of feature extraction because the neural network uses only the original input for learning weights rather than backpropagation, which has labels. They use unlabeled data in unsupervised learning and build a compressed representation of the input data. An autoencoder is trained to reproduce its own input data.

There are different types of autoencoders. Vanilla encoder is a three layered net, where the input and output are the same. If one hidden layer is not enough, we can obviously extend the autoencoder to use more hidden layers, known as multilayer autoencoder. Convolutional autoencoder uses images (3D vectors) instead of flattened 1D vectors. The input image is downsampled to give a latent representation of smaller dimensions and forces the autoencoder to learn a compressed version of the images. Regularized autoencoder uses a loss function that encourages the model to have other properties besides the ability to copy its input to its output.

3.2.2 Generative Adversarial Networks (GAN)

Generative Adversarial Networks (GAN) was first introduced by Ian Goodfellow and others from the University of Montreal in 2014. GANs are able to mimic any distribution of data in any domain: images, music, speech, and prose. GANs are an example of a network that use unsupervised learning to train two models in parallel. A key aspect of GANs (and generative models in general) is how they use a parameter count that is significantly smaller than normal with respect to the amount of data on which the network is trained. The network is forced to efficiently represent the training data, making it more effective at generating data similar to the training data. A GAN network is made up of a discriminator, *D*, and a generator, *G*, which operate in parallel. The generator's goal is to be able to create a fake output that resembles a real output, with the generator training through its interactions with the discriminator and not from any actual content [2]. The objective of the generator is to produce an output that is so close to real that it confuses the discriminator in being able to differentiate the fake data from the real data.

There are three steps in GANs. First, the generator takes in random numbers and returns an image. This generated image is fed into the discriminator alongside a stream of images taken from the actual dataset. Second, the discriminator takes in both real and fake images and returns probabilities; an output close to 0 being the data from the generator is fake and an output close to 1 being that the data is real. Third, the discriminator network provides feedback to the generator in order to train it and improve its output. GAN has the potential to be used in many applications and has been used in improving the resolution of images [22]. Another useful application using GAN has been the ability to create photos based on a detailed caption description, such as a caption stating "a yellow truck with white doors" used to generate a corresponding image [27] (Fig. 5).

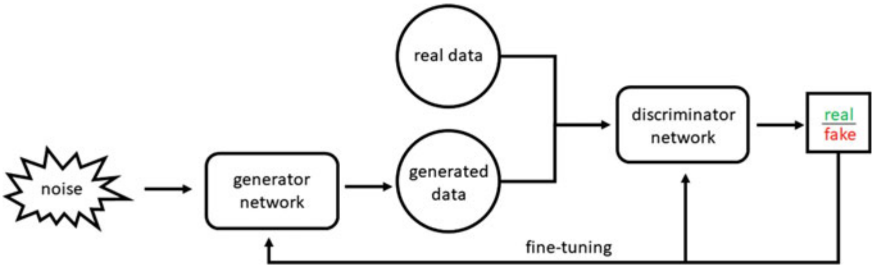


Fig. 5 A representation of a Generative Adversarial Network (GAN). It contains a generator network and a discriminator network which generator creates a dataset from random noise to feed into the discriminator to be able to differentiate the generated dataset from a real dataset

3.2.3 Deep Belief Network

After discussing the different machine learning networks and how they operate, we examine how these different networks have been used together. Neural networks can be joined together in different combinations in series with one another. In order to accomplish this, a link is established between each network. This is called as Deep Belief Network (DBN). It is structured by connecting multiple, smaller unsupervised neural networks, and forms an extensive layered connection. To understand the concept further, we have to dig deep into the components of a DBN: Belief Net and the Restricted Boltzmann Machine.

A Belief Net consists of randomly generated binary unit layers, where each of the connected layers have been assigned a weight function. The range of these binary units is from "0" to "1", and the probability of achieving the value "1" depends on the bias and weight factor inputs from the other connected units. Due to the layer-by-layer learning, we can determine how a variable present in one layer can possibly interact with those variables in another level. After the learning process, the values of variables can be effectively inferred by a bottom-up approach starting with a data vector in the bottom layer, and adding the generative weight function in the opposite direction.

A Restricted Boltzmann Machine (RBM) is a stochastic Recurrent Neural Network (RNN) consisting of randomly generated binary units, with undirected edges between the units. Since the major limitation of RBM is scalability, they are observed to have restricted connections between each of the hidden units.

3.3 Recurrent and Recursive Neural Networks

This class of deep learning structures has the ability to send data over time steps. We introduce 4 structures in this class: 1. Recurrent Neural Network, 2. Recursive Neural Network, 3. Long Short-term Memory (LSTM), 4. Attention.

3.3.1 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of deep learning based of the works of David Rumelhart in 1986. RNNs are heralded for their ability to process and obtain insights from sequential data. Therefore, video analysis, image captioning, natural language processing (NLP), and music analysis all depend on the capabilities of recurrent neural networks. Unlike standard neural networks that assume independence among data points, RNNs actively capture sequential and time dependencies between data.

One of the most defining attributes about RNNs is parameter sharing. Without parameter sharing, a model allocates unique parameters to represent each data point

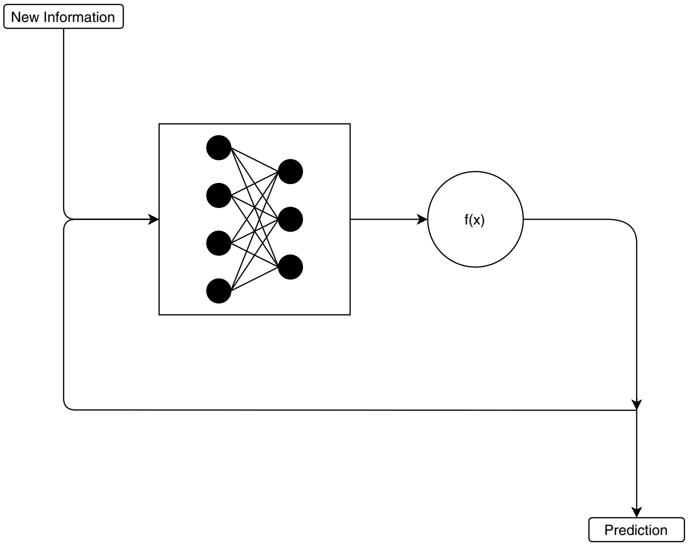


Fig. 6 A condensed representation of Recurrent Neural Network (RNN). It is a neural network that recurs over time, which allows information to persist by loops. The f(x) represents some squashing function

in a sequence and therefore cannot make inferences about variable length sequences. The impact of this limitation can be fully observed in natural language processing. For example, the sentences to decode are “Kobe Bryant is an incredible basketball player” and “An incredible basketball player is Kobe Bryant”. An ideal model should be able to recognize that ‘Kobe Bryant’ is the basketball player discussed in both sentences regardless the position of the words. A traditional multilayer network in this scenario would fail because it would create an interpretation of the language with respect to the unique weights set for each position (word) in the sentence. RNNs, however, would be more suitable for the task as they share weights across time steps (i.e. the words in our sentence)—enabling more accurate sentence comprehension [3] (Fig. 6).

RNNs generally augment the conventional multilayer network architecture with the addition of cycles that connect adjacent nodes or time steps. These cycles constitute the internal memory of the network which is used to evaluate the properties of the current data point at hand with respect to data points from the immediate past. It is also important to note that most conventional feedforward neural networks are limited to one to one mapping for input and output [3]. RNNs, however, can perform one to many, many to many (e.g. translating speech), and many to one (e.g. identifying voice) mappings. A computational graph is used to depict the mappings between inputs to outputs and loss. Unfolding the graph into a chain of events provides a clear picture of parameter sharing within the network. A generalized equation for recurrence relationships is

$$s^{(t)} = f(s^{t-1})) \tag{16}$$

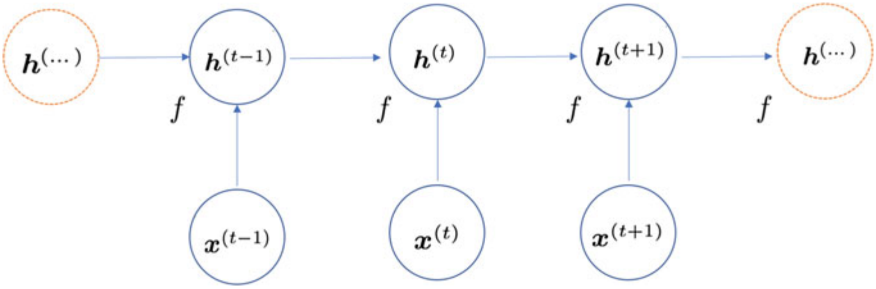


Fig. 7 An unfolded computational graph for RNN. Each node is associated with an instance of time

$s^{(t)}$ indicates the state of the system which is dependent on a previous time-step indicated by $t - 1$. This equation can then be re-written as

$$h^{(t)} = f(h^{t-1}), x^{^{(t)}};) \tag{17}$$

where $h^{(t)}$ is now used to represent the state and $x^{^{(t)}}$ denotes input from one particular time instance. The significance of $h^{(t)}$ is that it is a representation of the task-relevant aspects of the past sequence of inputs up to t [3] (Fig. 7).

Earlier versions of RNN architectures displayed great promise and versatility but were associated with certain notable flaws. RNN structures, in theory, are able to remember information for long periods of time, however, in practice, this is not always the case. Traditional RNN networks, also known as Vanilla RNNs, are especially prone to a vanishing gradient and an exploding gradient—both phenomenon resulting from propagation errors accumulated over many time steps. RNN works well in referencing bits of information if the gap between references remains small. Where RNN begins to suffer is when the gap between referenced data becomes large and RNN is not always able to make links between this data. Long Short-Term Memory (LSTM) and Truncated Backpropagation Through Time (TBPTT) are variants of traditional RNN architecture proposed to rectify these issues. LSTM architecture utilizes recurrent edges featuring fixed unit weights to counteract vanishing gradient. TBPTT architecture sets a cutoff for the number steps through which error can be propagated to rectify exploding gradient (Fig. 8).

Some other RNN architectures include Bidirectional Recurrent Neural Networks (BRNN) and Encoder-Decoder Recurrent Neural Networks (EDRNN). BRNNS deviate from the conventional causal structures utilized by most other RNN frameworks. They make inferences from the current data point in a sequence relative to both past and future data points. This is particularly useful for decoding the meaning of sentences in which each word of the sentence is evaluated in the context of all the values of the sentence. Furthermore, many subtle linguistic dependencies can be extrapolated by considering a word's left and right neighbors. It is also important to note that many words and phrases used in sentences can have different mean-

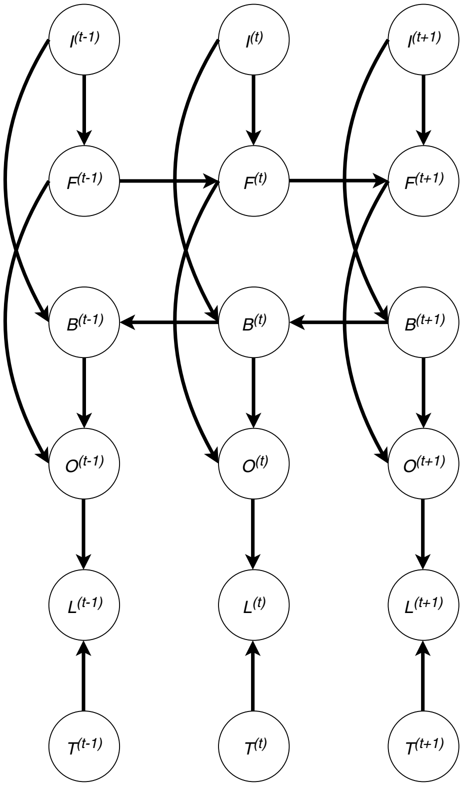


Fig. 8 Bidirectional RNN

ings depending upon the context of the sentence. A bidirectional view enables the model to have a higher probability of correctly extrapolating this context. In addition to NLP, BRNNs are also particularly useful in proteomics—identifying protein sequences from amino acid ordering—as well as in handwriting identification. EDRNN is another versatile RNN framework that allows the RNN to be trained to map an input sequence to variable length output sequences. This framework can be very useful to decode speech as well as to automate responses to speech.

3.3.2 Recursive Neural Network

Recursive neural networks, not to be confused with RNNs, are a set of non-linear adaptive models which are used to process data of variable length. They are especially proficient in processing data structure inputs. Recursive networks feed the state of the network back into itself, in what can be viewed as a loop. They are primarily suited for image and sentence deconstruction. The architecture of recursive neural networks enables users to not only identify the constituents of input data but also to quantitatively determine the relationships between them [3]. This kind

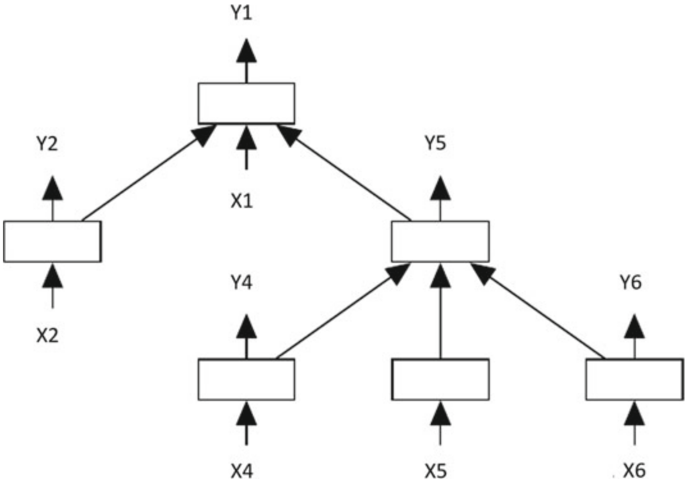


Fig. 9 A condensed representation of Recursive Neural Network

of deconstruction is made possible through a shared-weight matrix and binary tree structure—both of which enable the recursive neural network to extrapolate from varying length sequences of images and words. Furthermore, one major advantage of recursive nets over recurrent nets is that for a sequence of the same length n the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from n to $log(n)$ which enables efficient capturing of long-term dependencies [3]. Recursive neural networks are generally known for having a bottom-up feed-forward method and top-down propagation method. Both mechanisms constitute the propagation through structure that is prevalent in most recursive networks (Fig. 9).

Two of the most commonly used varieties of recursive networks include the semi-supervised recursive autoencoder and the supervised recursive neural tensor. The recursive autoencoder is used to deconstruct sentences for NLP applications whereas the recursive neural tensor is primarily used for computer vision applications. One drawback common to nearly all recursive neural networks is substantial computational overhead—moreso than recurrent neural networks. Recursive networks are reputed for processing exorbitant amounts of data often containing millions of parameters which results in long training times. As a result, optimization techniques are continuously developed for these architectures; furthermore, the ever-growing sophistication of processors and advancements made in parallel computing enable large-scale deployment of recursive neural networks.

3.3.3 LSTM

LSTM is the most common RNN architecture that remembers values over arbitrary intervals. It was first introduced in 1997 by Hochreiter and Schmidhuber and works well on making predictions based on time series data, avoiding the long-term dependency problem that traditional, or vanilla, RNNs were plagued with. LSTM is also well suited to classification and processing tasks and can be found in the Google Translate, Apple Siri, and Amazon Alexa applications.

As previously mentioned, RNN suffers from a context problem which is attributable to the phenomenon known as the vanishing gradient problem. The vanishing gradient problem occurs when gradient descent is used as an optimization algorithm along with backpropagation [5]. As gap sizes increase between dependencies, the error gradients vanish exponentially and may result in the training of a network to become very slow or even unable to learn.

With stochastic gradient descent, the gradient is calculated based on the partial derivative of the loss function with respect to the weights using backpropagation using the chain rule [1]. Stochastic gradient descent is an optimized form of gradient descent. While gradient descent optimizes the loss of all training samples in the network, it is intensive because it optimizes the loss for each training sample; if there are one million training samples, then the gradient is calculated one million times. Using stochastic gradient descent, only one training sample is used to optimize

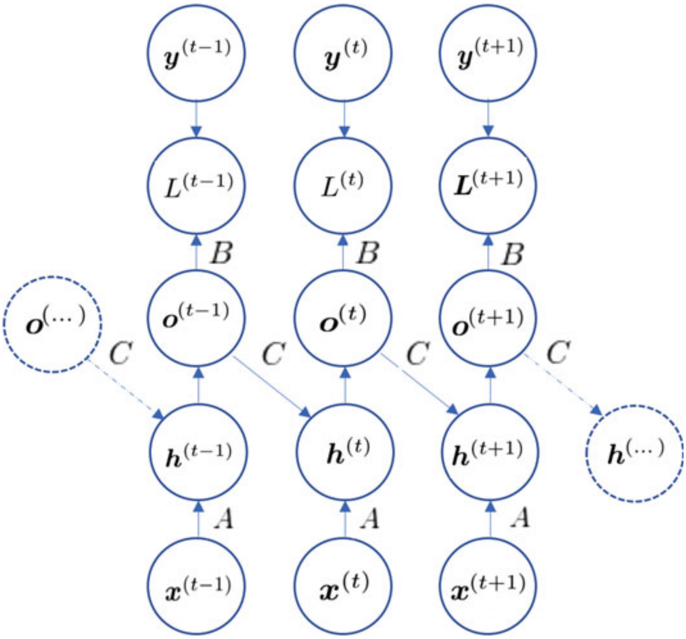


Fig. 10 An expansive computational graph for RNN

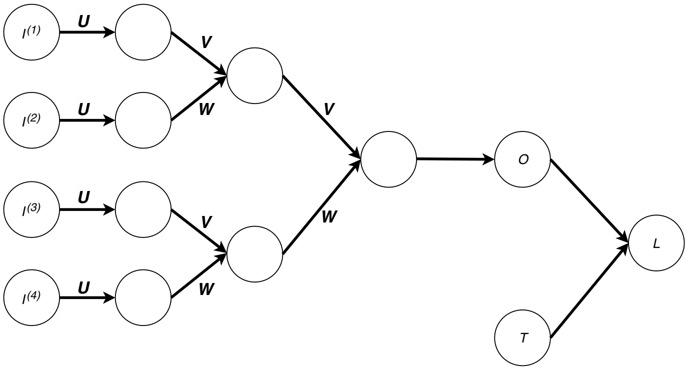


Fig. 11 RNN chain to binary tree structure enables the recursive neural network to extrapolate from varying length sequences of images and words

the parameters of the network, drastically reducing the time to train the network [23]. Additionally, another method, minibatch gradient descent can also be used to optimize the cost. Minibatch gradient descent instead uses n number of samples to update the parameters throughout the network. Although stochastic gradient descent will not reach the maximum optimization as compared with gradient descent, in general, the accuracy is sufficiently close to the optimization and is greatly beneficial when training a network with a large dataset [23] (Fig. 10).

Updates to the parameters in the network are applied using the chain rule. With the chain rule, the gradients are calculated as the product of the derivative of the cost function with respect to the weight from each node as it propagates back up the chain toward the front of the network. The gradient is then used to update the weights of the functions from earlier nodes. As the time dependency between layers increases, the weights are only marginally updated due to “vanishingly” small calculated corrections to the weight [5] (Fig. 11).

Consider the calculated gradient with a value less than one; with backpropagation, the weights are adjusted backward and if the gradients contain many small numbers less than one, then the gradient becomes exponentially small the further back the network it propagates; they become even smaller once multiplied by the learning rates. Since weights are initially set to an arbitrary number when setting up a network for training, they tend to initially have greater loss, compounding the problem of the vanishing gradient problem since the weights are only marginally adjusted. LSTM then addresses this problem by the use of different gates within its cell structures [5].

Different from classical RNN networks, LSTM not only can derive the information from the current state but it can also acquire information from previous states [11]. The LSTM models are used as follows,

$$f_t = \sigma_g(W_f y_t + U_f h_{t-1} + \beta_f) \tag{18}$$

$$i_t = \sigma_g(W_i y_t + U_i h_{t-1} + \beta_i) \tag{19}$$

$$o_t = \sigma_g(W_o y_t + U_i h_{o-1} + \beta_o) \tag{20}$$

$$h_t = o_{_t} \circ \sigma_h(c_{_t}) \tag{21}$$

$$c_t = f_{_t} \circ \sigma_{t-1} + i_{_t} \circ \sigma_c(W_c y_t + U_i h_{c-1} + \beta_c) \tag{22}$$

where y_t is the input vector, h_t is the output vector, c_t is the cell state vector; W, U, and β are matrices and vector parameters; and f_t, i_t, and o_t are forget gate vector, input gate vector, and output gate vector, respectively [12].

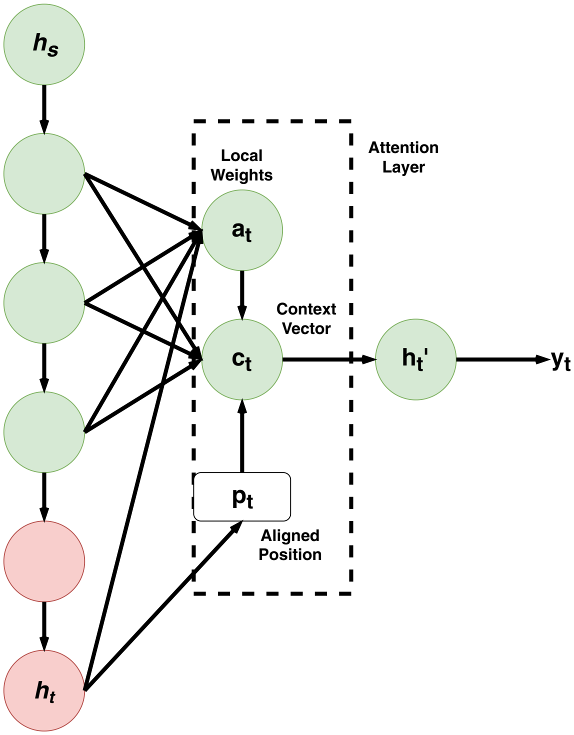
The critical components of the LSTM are the memory cell and its gates. There are different variations of LSTM but they all predominantly include three gates, known as the forget gate, input gate, and output gate. The contents of the memory cell are modulated by the input gates and forget gates. Assuming that both of these gates are closed, the contents of the memory cell will remain unmodified between one time-step and the next. The gating structure allows information to be retained across many time-steps, and consequently also allows gradients to flow across many time-steps. This allows the LSTM model to overcome the vanishing gradient problem that occurs with most Recurrent Neural Network models. The unfolded graph of an LSTM network can be thought of as a conveyor belt, with the data passing along the from one layer to the next, being altered slightly as it passes through each layer by use of the input and forget gates using linear interactions.

The forget gate is responsible for removing information from the cell state and its goal is to identify which information is no longer useful and may be forgotten. It takes 2 inputs: the Hidden State from the previous memory cell, $h_{(t-1)}$, and the Current Input, $x_{(t)}$, also known as the current cell state at that particular time step. The inputs are multiplied by weight matrices and a bias is added. After that, a sigmoid function is applied; the sigmoid function is responsible for deciding which values to keep and which to discard. The function outputs a vector with values 0 to 1; a 0 indicates the forget gate wants to forget the information completely while a 1 indicates the forget gate wants to remember the entire piece of information.

The input gate involves a 2-step process and is responsible for deciding what new information will be added to the cell state. Similar to the forget gate, a sigmoid function is applied to $h_{(t-1)}$ and $x_{(t)}$. A hyperpolic tangent function creates a vector of all possible values, ranging from -1 to 1. This vector indicates candidate values which may be added to the cell state.

The output gate selects useful information from the cell state as output in a 3-step process. In the first step, a hyperbolic tangent function is applied to cell state, creating a vector with scaled values from -1 to 1. Step 2 is to use sigmoid function and use the previous hidden state, $h_{(t-1)}$, and $x_{(t)}$ as inputs to create a regulatory filter. In the final step, the regulatory filter from step 2 is multiplied with the vector from step 1, producing an output and hidden state to the next cell. Using LSTM, the network is able to minimize any long term dependencies and can bridge gaps in data references in excess of 1,000 steps [5] (Fig. 12).

Fig. 12 Attention network



3.3.4 Attention

Most contemporary neural network architectures utilize recurrence and convolution mechanisms along with an encoder-decoder configuration. Attention networks use an additional "attention" mechanism that is growing in popularity among numerous architectures. Attention can be thought of similarly to how we focus our attention on the task at hand. For example, if you are asked to fix paint a room, you put your attention to the area of the room you are currently painting. If you are asked to fix a damaged vehicle, then your attention is on the part of the vehicle you are currently working on. Attention networks apply the same concept by focusing on specific areas at different time steps.

Using attention, models display higher prediction accuracy by finding global dependencies between data points without regard to their distance in both input and output sequences [30]. In addition to this benefit, attention mechanisms also make computations performed by the neural network more parallelizable. Attention mechanisms are generally used in conjunction with recurrence and convolution; in a small fraction of neural network architectures, attention may entirely replace recurrence and convolution schemes. Vaswani et al. have implemented such a novel architecture devoid of recurrence known as Transformer. This architecture makes use of an attention scheme called self-attention or intra-attention, in which numerous relationships are extrapolated between different positions of a data sequence [30]. Thus, by finding more patterns from input data, the Transformer's attention mechanism allows for more robust model creation.

4 Conclusions

The incorporation of deep learning models have allowed for large amounts of data to be correlated from multiple modalities. Built to emulate the structure of synaptic connections in the human brain, deep learning architectures are ubiquitously used for feature extraction, pattern analysis, and data abstraction. These models have been shown to perform better and faster than current state-of-the-art analysis techniques through supervised, unsupervised, and semi-supervised learning tasks. This chapter has reviewed numerous common structures that have developed recently in the following three classes.

1. The convolutional neural network which is a powerful deep learning method inspired by biological processes. It uses little pre-processing compared to other machine learning algorithms, where traditional algorithms need hand-engineered filters to preprocess data. It has a variety of applications from image recognition to natural language processing, with a multi-layer structure containing convolution layers, pooling layers, and fully connected layers.

2. The pretrained unsupervised networks which are a class of deep learning algorithms which generate data and extract features. The main architectures of PUN includes Autoencoders, Generative Adversarial Networks, and Deep Belief Networks. Using these architectures, we could generate similar data from the original dataset to improve model accuracy.

3. The recurrent/recursive neural networks which are a class of deep learning structures that have the ability to send data over time steps. We introduced 4 structures in this class, including Recurrent Neural Network, Recursive Neural Network, Long Short-term Memory, and Attention.

There is a large range of applications that deep learning algorithms could be used for. They can be used to perform classification, data generation, and information understanding. For various fields from autonomous driving to bioinformatics, and medical image processing to assist the medical field in making accurate diagnoses [8, 16, 21]. For example, many CNN architectures are developed for image recognition tasks, including AlexNet and GoogLeNet. LSTM architectures have been designed for natural language processing since they have shown high performance in this application [28]. A CNN-based architecture called AtomNet [31] is designed for drug discovery and successfully predicted some novel molecules for Ebola virus Fig. 13. Deep and thorough researches has been done with using different deep learning architectures to analyze multimodality in medical imaging techniques [12].

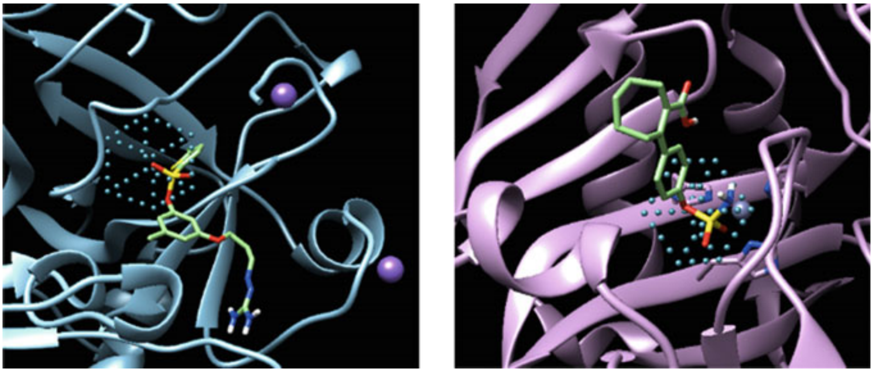


Fig. 13 AtomNet is trained to recognize sulfonyl groups—a structure often found in antibiotics [31]

Financial service companies developed deep learning algorithms to detect fraud and prevent money laundering. The applications for deep learning expands every month to every corner of academia and industry.

References

1. Boden, M.: A guide to recurrent neural networks and backpropagation. The Dallas project (2002)
2. Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., Bharath, A.A.: Generative adversarial networks: an overview. IEEE Signal Process. Mag. **35**(1), 53–65 (2018)
3. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press. http://www.deeplearningbook.org (2016)
4. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
5. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
6. Hosseini, M., Pompili, D., Elisevich, K., Soltanian-Zadeh, H.: Optimized deep learning for EEG big data and seizure prediction BCI via internet of things. IEEE Trans. Big Data **3**(4), 392–404 (2017)
7. Hosseini, M.-P.: Developing a cloud based platform as a service to improve public health of epileptic patients in urban places. Reimagining Health in Cities: New Directions in Urban Health Research, Drexel University School of Public Health, Philadelphia, USA (2015)
8. Hosseini, M.-P.: Proposing a new artificial intelligent system for automatic detection of epileptic seizures. J. Neurol. Disorders **3**(4) (2015)
9. Hosseini, M.-P.: A cloud-based brain computer interface to analyze medical big data for epileptic seizure detection. In: The 3rd Annual New Jersey Big Data Alliance (NJBDA) Symposium (2016)
10. Hosseini, M.P.: Brain-computer interface for analyzing epileptic big data. Ph.D. thesis, Rutgers University-School of Graduate Studies (2018)

11. Hosseini, M.-P., Hajisami, A., Pompili, D.: Real-time epileptic seizure detection from EEG signals via random subspace ensemble learning. In: 2016 IEEE International Conference on Autonomic Computing (ICAC), pp. 209–218. IEEE (2016)
12. Hosseini, M.P., Lau, A., Lu, S., Phoa, A.: Deep learning in medical imaging, a review. IEEE Rev. Biomed. Eng. (2019)
13. Hosseini, M.-P., Pompili, D., Elisevich, K., Soltanian-Zadeh, H.: Random ensemble learning for EEG classification. Artif. Intell. Med. **84**, 146–158 (2018)
14. Hosseini, M.P., Soltanian-Zadeh, H., Akhlaghpoor, S.: Three cuts method for identification of COPD. Acta Medica Iranica 771–778 (2013)
15. Hosseini, M.-P., Soltanian-Zadeh, H., Elisevich, K., Pompili, D.: Cloud-based deep learning of big EEG data for epileptic seizure prediction. In: 2016 IEEE Global Conference on Signal and Information Processing (GlobalSIP), pp. 1151–1155. IEEE (2016)
16. Hosseini, M.-P., Tran, T.X., Pompili, D., Elisevich, K., Soltanian-Zadeh, H.: Deep learning with edge computing for localization of epileptogenicity using multimodal rs-fMRI and EEG big data. In: 2017 IEEE International Conference on Autonomic Computing (ICAC), pp. 83–92. IEEE (2017)
17. Karnin, E.D.: A simple procedure for pruning back-propagation trained neural networks. IEEE Trans. Neural Netw. **1**(2), 239–242 (1990)
18. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
19. Le Cun, Y., Jackel, L.D., Boser, B., Denker, J.S., Graf, H.P., Guyon, I., Henderson, D., Howard, R.E., Hubbard, W.: Handwritten digit recognition: applications of neural network chips and automatic learning. IEEE Commun. Mag. **27**(11), 41–46 (1989)
20. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
21. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436 (2015)
22. Ledig, C., Theis, L., Huszár, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., et al.: Photo-realistic single image super-resolution using a generative adversarial network. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4681–4690 (2017)
23. Liang, N.-Y., Huang, G.-B., Saratchandran, P., Sundararajan, N.: A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Trans. Neural Netw. **17**(6), 1411–1423 (2006)
24. Liao, D., Lu, H.: Classify autism and control based on deep learning and community structure on resting-state fMRI. In: 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), pp. 289–294. IEEE (2018)
25. Patterson, J., Gibson, A.: Deep Learning: A Practitioner's Approach. O'Reilly Media, Inc. (2017)
26. Puskorius, G., Feldkamp, L.. Truncated backpropagation through time and kalman filter training for neurocontrol. In: Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94), vol. 4, pp. 2488–2493. IEEE (1994)
27. Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., Lee, H.: Generative adversarial text to image synthesis (2016). arXiv preprint arXiv:1605.05396
28. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
29. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Computer Vision and Pattern Recognition (CVPR) (2015)
30. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)
31. Wallach, I., Dzamba, M., Heifets, A.: AtomNet: a deep convolutional neural network for bioactivity prediction in structure-based drug discovery (2015). arXiv preprint arXiv:1510.02855

Theoretical Characterization of Deep Neural Networks

Piyush Kaul and Brejesh Lall

Abstract Deep neural networks are poorly understood mathematically, however there has been a lot of recent work focusing on analyzing and understanding their success in a variety of pattern recognition tasks. We describe some of the mathematical techniques used for characterization of neural networks in terms of complexity of classification or regression task assigned, or based on functions learned, and try to relate this to architecture choices for neural networks. We explain some of the measurable quantifiers that can been used for defining expressivity of neural network including using homological complexity and curvature. We also describe neural networks from the viewpoints of scattering transforms and share some of the mathematical and intuitive justifications for those. We finally share a technique for visualizing and analyzing neural networks based on concept of Riemann curvature.

Keywords Deep neural networks · Machine learning · Expressivity · Algebraic topology · Betti numbers · Curvature · Riemannian geometry · Scattering transform

1 Overview

Deep neural networks (DNNs), including CNNs, RNNs, GANS and other variants are the best performing machine learning algorithms for a broad range of varied pattern recognition tasks, including classification, object detection, semantic segmentation, speech recognition etc [1, 22, 23, 36, 39, 41]. The mathematical understanding of neural network of more than one hidden layer is still rather limited, due to inadequacy of mathematical models in effectively modeling the complex and non-linear hierarchical structures. This diminishes the ability of engineers to improve and customize

P. Kaul (✉) · B. Lall
Electrical Engineering Department, Indian Institute of Technology, Delhi, India
e-mail: eez157544@iitd.ac.in

B. Lall
e-mail: brejesh@iitd.ac.in

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_2

the networks in predictable manner. Though back-propagation algorithm and gradient descent based algorithms are widely effective, the optimizability and generalizability of various network architectures is poorly understood. Hence some architectures fail to converge to a good generalized solution, whereas others with slightly different layers converge well [18].

In this chapter, we review some research areas which utilize techniques from signal processing theory and differential topology [6, 11, 26, 33] to develop methods and techniques to better analyze, understand and visualize the effectiveness of DNNs in varied pattern recognition tasks. These techniques can also be extended for introducing algorithmic and architectural improvements in neural network architectures, including those for processing both Euclidean and non-Euclidean input data.

We start by describing in brief some mathematical concepts in topology theory and Riemannian geometry that are prerequisites to understand rest of this chapter. We also give a brief overview of upcoming field of geometric deep learning which uses graph signal processing and other ideas to extend deep learning to non-Euclidean data like graphs and manifolds [7, 40]. We do this in brief and without any proof. Next we describe the concept of expressive power and currently known bounds on the Vapnik–Chervonenkis dimensions [13, 21, 43] (VC dimension) of neural networks based on concepts of algebraic topology, especially using Betti numbers [3–5]. We also discuss some theoretical and empirical results for the same, which may have practical implication in neural network architecture design [17].

Next we describe the approach of exploring neural nets by analyzing simplified models using scattering transforms [2, 9, 28, 29, 44], which consist of wavelet transform layers interleaved with non-linearities and model their invariance to distortions in data as diffeomorphisms utilizing the theory of Lie groups and topological manifolds [14].

Lastly we discuss some very recent advances in the mathematical understanding of expressivity of deep neural networks utilizing Riemannian geometry. Some analytical and empirical results are also shared to show the variation in curvature on Riemannian manifolds by propagating data through the neural network. We explain the research on modeling deep learning networks by direct calculation of Riemannian and Ricci curvature tensors and how this can be used to characterize, analyze and even possibly aid in design of deep neural networks [34, 35, 37].

2 Neural Net Architecture

Though the details of neural network architecture and components are described elsewhere and not focus of this chapter, we give a short overview of the components to be able to define the terminology and mathematical representation for the basic building blocks. Neural networks are computation structures with learnable parameters which can be characterized as having multiple computational layers with layers connected to each other through directed graphs. Deep neural networks, used to designate neural networks with more than one hidden layers, are not well understood mathematically.

Neural Networks take a vector or tensor as input, which is then sequentially fed through a series of processing layers. The layers are of various types depending on the type of networks and the application. The layers include fully connected layers, convolution layers, pooling layers, non-linear activations, normalization layers, loss layers, etc. Multi layer perceptrons (MLPs) consist only of fully connected layers interleaved with non-linear activations. Convolutional Neural Networks (CNNs) have convolutional layers instead of fully connected layers in the initial phase of the network. Recurrent neural networks (RNNs) are distinguished from other classes by presence of components with memory e.g. long short term memory (LSTMs) and gated recurrent units (GRU) [10, 19]. These networks can be used for classification/regression of time series oriented data. Modern neural networks, especially CNNs, can be very deep with hundreds of layers.

Multi layer perceptrons (MLP) are the simplest form of neural networks consisting of fully connected layers couple with non-linearities (as shown in Fig. 1). The fully connected layers can be modeled as matrix to vector multiplies. The non-linearities are typically element-wise non-linear operations.

The CNNs consist of directed acyclic graphs (DAG) with nodes performing one of the operations described above e.g. convolution, pooling etc. The interconnections can be either parallel or serial or hierarchical (network within networks).In its simplest form, CNNs have a set of convolutional layers alternating with rectified linear units (ReLU) [15] layers in series (see Fig. 2). The CNNs are distinguished from the MLPs by sharing of weights across spatial dimensions. The purpose of weight sharing in spatial dimensions is to provide spatial invariance, which is a desired characteristic when input are images. Hence convolutional neural networks are ideally suited for object classification and detection networks working on image or video datasets, which requires the property of spatial invariance. The spatial dimension is reduced by introducing pooling layers, which will reduce the size of spatial dimensions. The pooling also helps with introducing translational and rotational invariance (or covariance) in the network. The final stages in convolutional networks may also include one or more fully connected layers which are equivalent to convolutional layers with filter of spatial size 1×1 and input of spatial size 1×1.

RNNs are family of neural networks that are used to process sequences. Analogously to CNNs which share weight across spatial dimensions for images, the RNNs share weights across temporal dimension in sequences. The basic building block consists of recurrent units, which have feedback connections from the output to input and maintain state (memory). There are many types of recurrent units, but the most common ones are gated units like long short term memory (LSTM as shown in Fig. 3 and RNN using LSTM is shown in Fig. 4) and gated recurrent units (GRU). Besides, fully connected multipliers and non-linearities, the ability to selectively learn, forget and output the internal state based on learned coefficients is a feature of these gated units, which provides them ability to keep a memory for sufficiently long periods of time as required. The coefficients of gates are learned in the training processes.

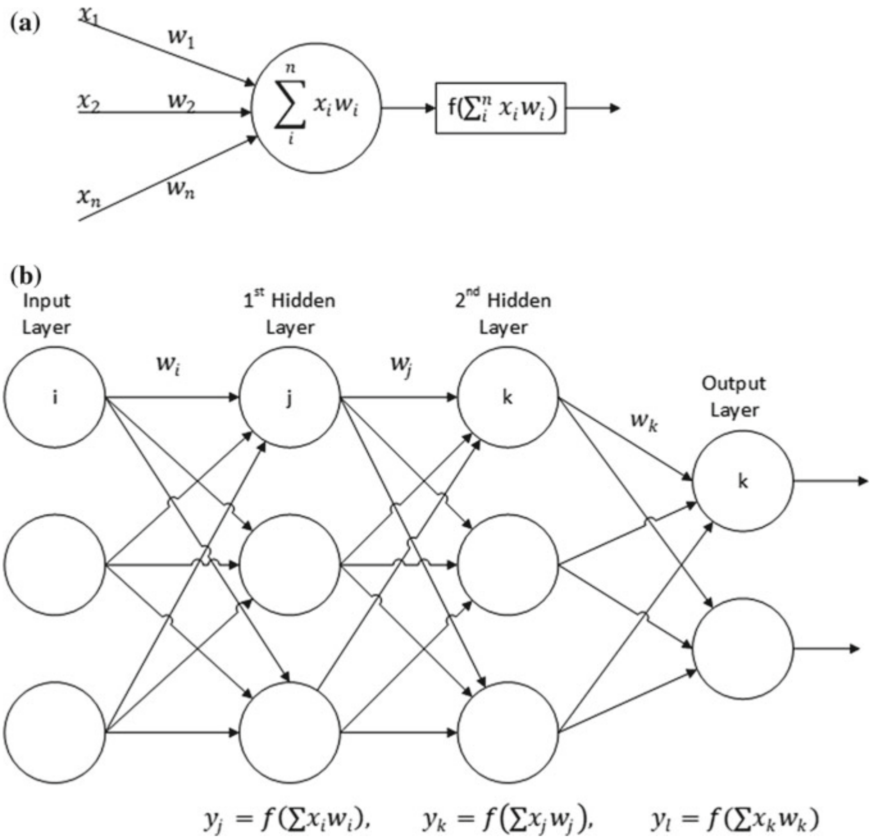


Fig. 1 Multi layer perceptrons. **a** a single neuron. **b** MLP with 2 hidden layers

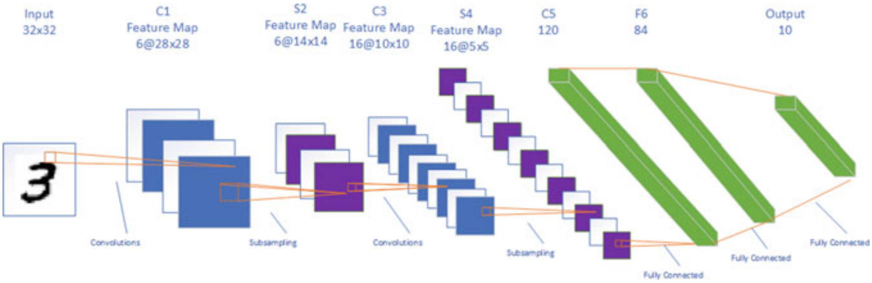


Fig. 2 Example of a CNN (LeNet-5 [24])

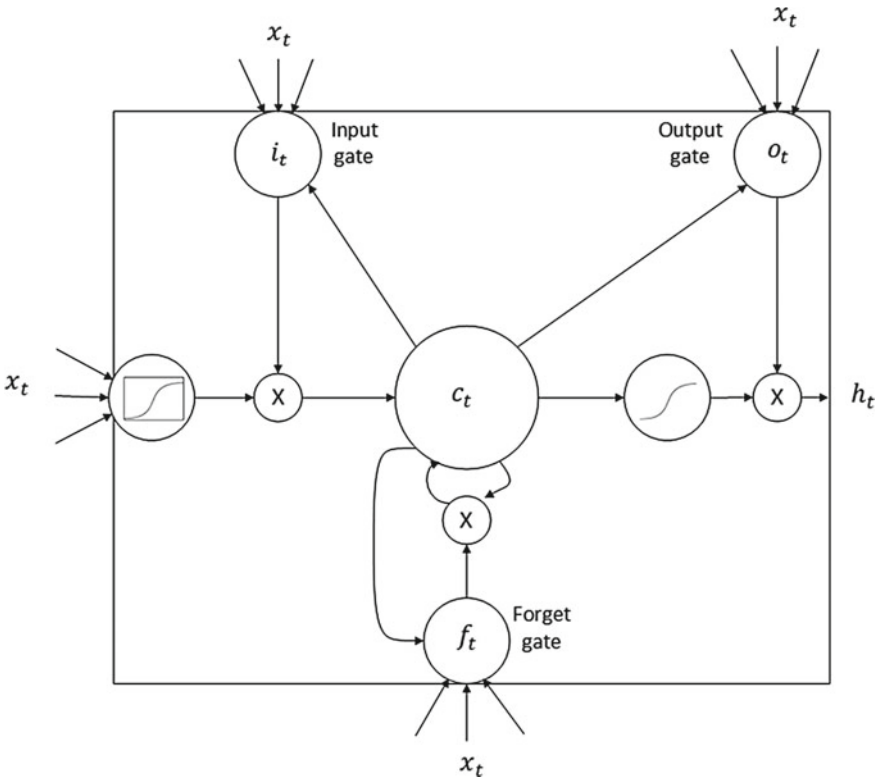


Fig. 3 LSTM unit

We can model fully connected layers as matrix of coefficients to input vector multiplication followed by point-wise nonlinearity. For CNNs, utilizing the im2col [30] type input expansion can be used to convert the input feature map of any convolutional layer to a 2D matrix. Hence we can represent convolution as a product between a matrix and a vector. Let the filter matrix be represented as F, and ϕ be the expansion operator (im2col), the output of convolutional layer can be represented as follows:

$$vec(y) = vec(x^{l+1}) = vec(\phi(x^l)F). \tag{1}$$

For a succession of layers with alternating convolutions and point-wise operator ρ given by $max(x, 0)$, we can represent the networks as

$$vec(y) = vec(...vec(\phi(vec(\phi(x^1)F_1))F_2)...F_J). \tag{2}$$

For more complex directed acyclic graphs (DAG) the same expression can be extended. The neural networks are trained by minimizing a structural risk function which includes a term used to measure the error between the prediction from the neural network represented by $y^{(i)}$ and the ground truth $(\hat{y})^{(i)}$. Generally, the structural risk function of a model consists of a empirical risk term and a regularization term, which can be represented as

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \boldsymbol{\mathcal{L}}(\boldsymbol{\theta}) + \lambda \cdot \Phi(\boldsymbol{\theta}) \tag{3}$$

$$= \arg\min_{\boldsymbol{\theta}} \frac{1}{n} \sum_{i=1}^{n} L\big(y^{(i)}, \hat{y}^{(i)}\big) + \lambda \cdot \Phi(\boldsymbol{\theta}) \tag{4}$$

where $\Phi(\theta)$ is the regularization or penalty term, and θ represents the parameters of the model to be learned. There are many types of loss functions like mean squared error (MSE), cross entropy, Kullback–Leibler divergence [12, 16] etc. A example of a simple square l_2 loss function can be the following:

$$\boldsymbol{\mathcal{L}} = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2. \tag{5}$$

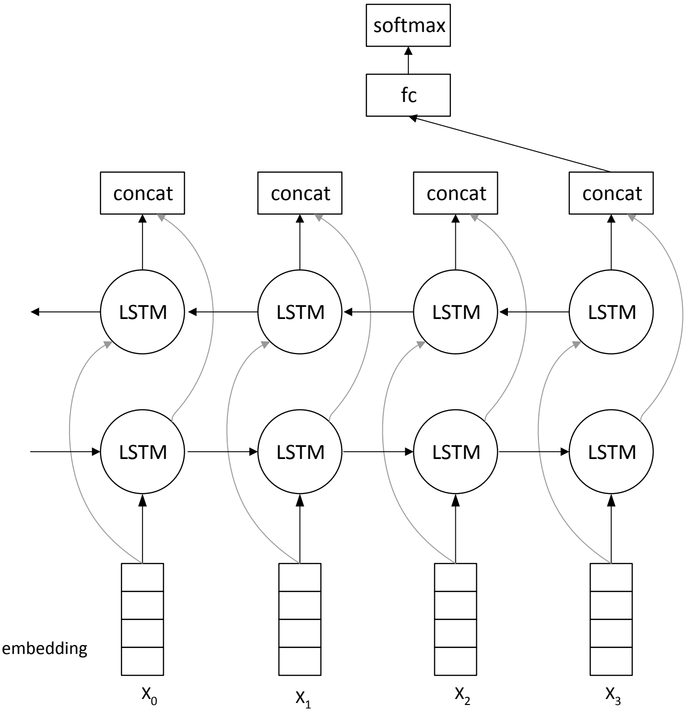


Fig. 4 Recurrent neural networks

3 Brief Mathematical Background

3.1 Topology and Manifolds

Topology is a branch of mathematics which deals with characterizing shapes, space and sets by their connectivity. In topology, we express relationship between two spaces through continuous maps between them.

Definition 3.1 Let M be a set and $\mathcal{P}(M)$ be the set of all subsets of M (i.e. power set of M).
A set $\mathcal{O} \subseteq \mathcal{P}(M)$ is called a **topology**, if all of the following properties are satisfied:

(i) $\mathcal{O}$ contains the null set and the set M itself.
(ii) any union of subsets of $\mathcal{O}$ is contained in $\mathcal{O}$, or more formally, $A \in \mathcal{O},\ \ B \in \mathcal{O} \implies A \cap B \in \mathcal{O}$.
(iii) lastly, any intersection of finite number of subsets of $\mathcal{O}$ is contained in $\mathcal{O}$, i.e., $U_\alpha \in \mathcal{O},\ \ \alpha \in \mathcal{I}$ ($\mathcal{I}$ is an index set) $\implies \left(\bigcup_{\alpha\in\mathcal{I}} U_\alpha\right) \in \mathcal{O}$.

The sets in $\mathcal{O}$ are called open sets.

The notion of topological equivalence (called homeomorphism) implies that there exists a continuous map $f : A \mapsto B$ for which the inverse function f^{-1} is also continuous.

Definition 3.2 A function $f : X \mapsto Y$ between two topological spaces (X, T_X) and (Y, T_Y) is called a homeomorphism if all of the following properties are satisfied:

- f is a both a one-to-one and onto mapping,
- f is continuous,
- the inverse function of f is also continuous.

A **neighborhood** of a point x in M is a set $N(x)$ containing an open set which contains the point x. A family of neighborhoods of x implies a set of points that are "near to x". A topological space is called **Hausdorff (separated)** if for any two distinct points there always exist disjoint neighborhoods.

Definition 3.3 A paracompact Hausdorff topological space $(M, \mathcal{O})$ is called a **d-dimensional topological manifold** if $\forall p \in M : \exists U \in \mathcal{O} : p \in U, \exists$ homeomorphism $x : U \to x(U) \subseteq \mathcal{R}^d$ satisfying the following:

(i) x **is invertible**: $x^{-1} : x(U) \to U$,
(ii) x **is continuous** w.r.t. $(M, \mathcal{O})$ and $(\mathcal{R}^d, \mathcal{O}_{std})$,
(iii) x^{-1} **is continuous**.

A d-dimensional topological manifold is locally homeomorphic to d-dimensional euclidean space (R^n) at every point. So at every point in the manifold there exists a mapping which maps an open set on the manifold to a part of the Euclidean space. Such a mapping is called a chart. The set of such overlapping charts which cover the entire manifold is called an atlas. Two overlapping charts are shown in Fig. 5.

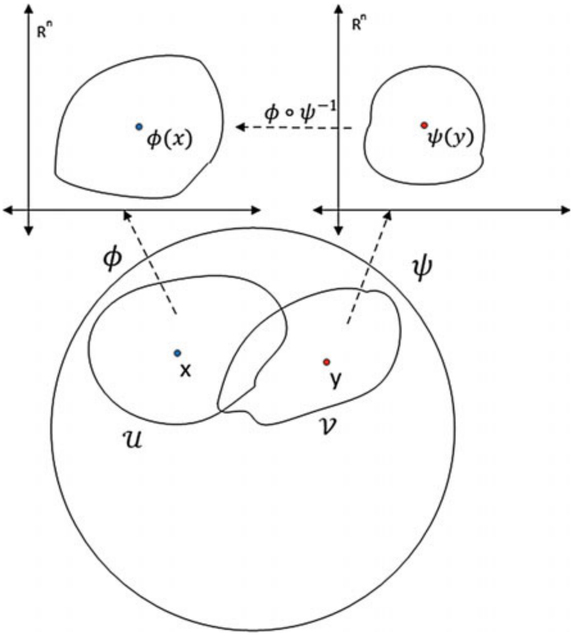


Fig. 5 Manifold. The two regions $\mathcal{U}$ and $\mathcal{V}$ on the manifold map to two different maps $\phi(x)$ and $\psi(x)$ with some overlap

Definition 3.4 A curve on a manifold M is a smooth (i.e. C^{∞}) map from some open interval $(-\epsilon, \epsilon)$ of a real line onto M.

Two curves $\sigma_1(0)$ and $\sigma_2(0)$ are tangent at a point p in M if $\sigma_1(0) = \sigma_2(0) = p$ and in some local coordinate system they are tangent in the usual sense of curves in $\mathcal{R}$.

A **tangent vector** $p \in M$ is a equivalence class of curves in M where the equivalence relation is that the two curves will be tangent at point p. We write the equivalence class of a particular curve σ as $[\sigma]$ (see Fig. 6).

A function with the three properties defined in Definition 3.2 is called bi-continuous. If a bi-continuous function exists, we say X and Y are homeomorphic. For topological spaces, homeomorphisms form an equivalence relation. The resulting equivalence classes are called homeomorphism classes. In case of smooth manifolds, topological equivalence (homeomorphism) which retains smoothness is called diffeomorphism.

Algebraic Topology assigns algebraic objects like groups, chains and similar objects to topological spaces. Two spaces can be thought of a topologically equivalent if the algebraic objects to which they are assigned are isomorphic. In the context of characterizing neural networks, characterization can be done by topological connectivity of the dataset. Then the expressivity of a particular neural network is assessed to be their capacity to produce decision regions with the same connectivity. This aspect is explained in more detail in Sect. 4.

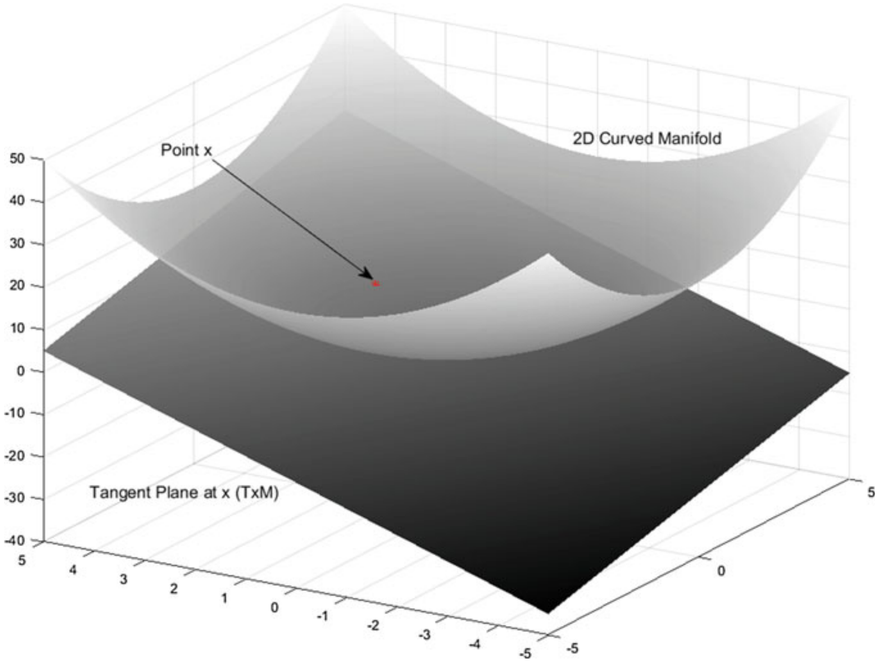


Fig. 6 Tangent space $T_x M$ of a single point, x, on a manifold

3.2 Riemannian Geometry and Curvature

A Riemannian manifold is any smooth manifold over which a symmetric tensor of form $\binom{0}{2}$ is defined. Such a tensor **g** is called a metric. If the metric is not positive definite, then such geometry is called pseudo-Riemannian. At each location on the manifold, such a metric gives a mapping between vectors spaces and their duals (one-forms). A vector field is a vector valued function which assigns a vector to any point on a manifold. For any vector field $\boldsymbol{A}(x)$ on a point x, there is a mapping given by the metric to the dual vector field:

$$\tilde{A} = g(\boldsymbol{A}, \cdot). \tag{6}$$

The dual vector field $\tilde{A}$ can be thought of as acting on $\boldsymbol{B}$ which is equivalent to a standard dot-product. i.e.,

$$g(\boldsymbol{A}, \boldsymbol{B}) = \tilde{A}(\boldsymbol{B}) = \boldsymbol{A} \cdot \boldsymbol{B}. \tag{7}$$

Einstein convention for summations is used in most of mathematical physics literature involving relativity and gravitation. The same has been used here and will be explained now. In this convention, indices are represented by subscripts and superscripts. Subscripts indicate contra-variant vectors and subscripts indicate covariant vectors, in case those are included only once in the equation. In case a equation has indices repeated both as top and bottom indices on separate terms, those can be contracted i.e. thought as part of a summation, and the summation sign can be omitted.

The derivatives of the fields are also represented in a shortened notation, with a comma before the subscript variable indicating derivative is w.r.t. that variable:

$$\frac{\partial \psi}{\partial \alpha} = \psi_{,\alpha} \,. \tag{8}$$

The co-vectors have gradients which are defined as:

$$(\tilde{\partial} \psi) = (\frac{\partial \psi}{\partial \alpha_1}, \frac{\partial \psi}{\partial \alpha_2}, \frac{\partial \psi}{\partial \alpha_3}, \frac{\partial \psi}{\partial \alpha_4} \ldots). \tag{9}$$

Each of the components of the gradient of dual vector space can be given by a matrix transformation Λ_{α}^{β} as shown below:

$$(\tilde{\partial} \psi)_{\bar{\alpha}} = \Lambda_{\alpha}^{\beta} (\tilde{\partial \psi})_{\beta}, \tag{10}$$

The need for covariant derivatives and their relationship with standard derivatives is not immediately apparent. To understand this we need to note that basis vectors in case of curved co-ordinates can vary over space. To take the example of rectangular and polar co-ordinates the field $\boldsymbol{e}_z$, which is the unit basis vector in direction z, in Euclidean 3D space is constant across space. When we convert this to polar co-ordinates, this field becomes dependent on the co-ordinates and hence is varying across space.

$$\boldsymbol{e}_z = (\Lambda_z^r, \Lambda_z^{\phi}) = (cos\phi, -r^{-1} sin\phi), \tag{11}$$

In this case we do not get zero derivative by component wise differentiation of $\boldsymbol{e}_z$ w.r.t. ϕ, though we would expect the derivative to be zero even in curved coordinate system. Hence summation of derivatives of individual components of a vector is not equivalent to the derivative of the vector itself. For taking care of this inconsistency, covariant derivatives have to be introduced. Those are defined as follows:

$$\frac{\partial \boldsymbol{A}}{\partial z^{\beta}} = \frac{\partial}{\partial z^{\beta}} (A^{\alpha} \boldsymbol{e}_{\alpha}) \tag{12}$$

$$= \frac{\partial A^{\alpha}}{\partial z^{\beta}} \boldsymbol{e}_{\alpha} + A^{\alpha} \frac{\partial \boldsymbol{e}_{\alpha}}{\partial z^{\beta}}, \tag{13}$$

Here, the two terms constitute the partial derivatives of vector components and the derivative of the basis vectors in the new coordinate system, respectively. This equation is complete for but the sake of simplicity, we additionally define Christoffel symbols $\Gamma^{\mu}_{\alpha\beta}$ as follows:

$$\frac{\partial \boldsymbol{e}_{\gamma}}{\partial z^{\kappa}} = \Gamma^{\nu}_{\gamma\kappa}\boldsymbol{e}_{\nu}. \tag{14}$$

The Christoffel symbol on the right can be inferred to be the μth component of derivative of $\frac{\partial \boldsymbol{e}_{\gamma}}{\partial x^{\kappa}}$. We can consider γ the index for the basis being differentiated and κ the coordinate against with the differentiation is done. Covariant derivative in (12) is now:

$$(\nabla \boldsymbol{A})^{\gamma}_{\kappa} = A^{\gamma}_{;\kappa} = A^{\gamma}_{,\kappa} + A^{\nu}\Gamma^{\gamma}_{\nu\kappa}. \tag{15}$$

Please note that we are using the notation for differentiation indicated in in Eq. 8 above. Christoffel Symbols are calculated from metric as follows [38]

$$\Gamma^{\gamma}_{\beta\mu} = 1/2 * g^{\alpha\mu}(g_{\alpha\beta,\mu} + g_{\alpha\mu,\beta} - g_{\beta\mu,\alpha}). \tag{16}$$

By definition, parallel transport of a vector $\boldsymbol{A}$ on a path parameterized by scaler λ on any manifold is when the vector $\boldsymbol{A}$ is defined on all points on the path, and if the vectors on infinitesimally close points can be considered to be almost parallel, even if the vectors on further off points are not.

In Euclidean flat space, only straight lines parallel transport the tangent vector. In curved space too, we can find analogous paths to straight lines (called geodesics) by imposing the constraint that vectors on those get parallel transported along that path. The geodesics are defined as.

$$\frac{d}{d\lambda}\left(\frac{dx^{\alpha}}{d\lambda}\right) + \Gamma^{\alpha}_{\mu\beta}\frac{\partial x^{\mu}}{d\lambda}\frac{dx^{\beta}}{d\lambda} = 0. \tag{17}$$

By definition Riemannian curvature can be thought of as the deviation in a vector when we try to parallel transport it along a close loop in the given coordinate space. Such a loop is shown in Fig. 7. The sides of the loop consist of lines $x = a$, $x = a + \delta a$, $y = b$ and $y = b + \delta b$.

As the unit vector passes through the loop composed of points P, Q, R, S and reaches back to the original point P, we find the deviation of this vector at the end compared to at the starting position. This deviation is given by:

$$\delta V^{\alpha} = \delta a \delta b[\Gamma^{\alpha}_{\mu 1,2} - \Gamma^{\alpha}_{\nu 2,1} + \Gamma^{\alpha}_{\nu 2}\Gamma^{\alpha}_{\mu 1} - \Gamma^{\alpha}_{\nu 1}\Gamma^{\alpha}_{\mu 2}]V^{\mu}. \tag{18}$$

Riemann Curvature is derived from the above calculation as a tensor of the form $\binom{1}{3}$ as given below.

$$R^{\alpha}_{\beta\mu\nu} = \Gamma^{\alpha}_{\beta\nu,\mu} - \Gamma^{\alpha}_{\beta\mu,\nu} + \Gamma^{\alpha}_{\sigma\mu}\Gamma^{\alpha}_{\sigma\nu} - \Gamma^{\alpha}_{\sigma\nu}\Gamma^{\alpha}_{\beta\nu}. \tag{19}$$

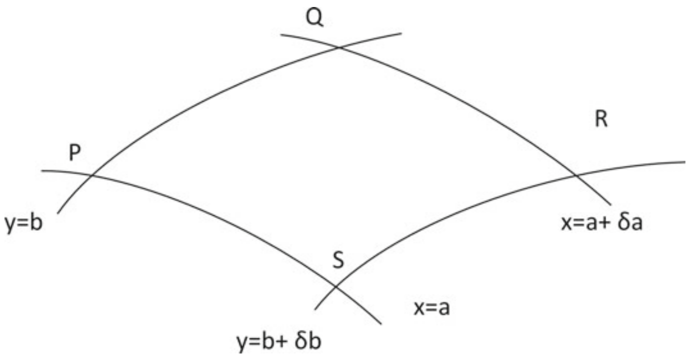


Fig. 7 Loop in section of co-ordinate grid

We can also calculate the Riemannian curvature directly from the metric instead of using Christoffel symbols.

$$R^{\alpha}_{\beta\mu\nu} = \frac{1}{2} g^{\alpha\sigma} (g_{\sigma\nu,\beta\mu} - g_{\sigma\mu,\beta\nu} + g_{\beta\mu,\sigma\nu} + g_{\beta\nu,\sigma\mu}). \tag{20}$$

To calculate Ricci tensor, we can contract Riemann curvature on first and third indices to give.

$$R_{\alpha\beta} = R^{\mu}_{\sigma\mu\beta} \tag{21}$$

Contracting Ricci tensor further gives the Ricci scaler.

$$R = g^{\mu\nu} R_{\mu\nu} = g^{\mu\nu} g^{\alpha\beta} R_{\alpha\mu\beta\nu}. \tag{22}$$

Curvatures metrics can be considered to be either intrinsic curvatures or extrinsic curvatures. Extrinsic curvature can be measured only when the manifold space is embedded in higher dimensions. For the simplest case of curves on planes, to measure the curvature on a point, we first find the osculating circle Fig. 8, which a circle that is the closest approximation of the curve at that point. The reciprocal of the radius of the osculating circle is considered to be the extrinsic curvature at that point. On the other hand we do not need embedding into a higher dimensional space in case of intrinsic curvature, which can by definition be measured from within the manifold space itself. The curvature tensors defined above are forms of extrinsic curvatures, namely Riemann and Ricci curvature tensors. The most basic form of extrinsic curvature is the Gaussian curvature. To find the Gaussian curvature around any point p in a manifold, we need to find the circumference C of a circle of radius ϵ drawn with the point as the center. Then Gaussian curvature can be calculated to be:

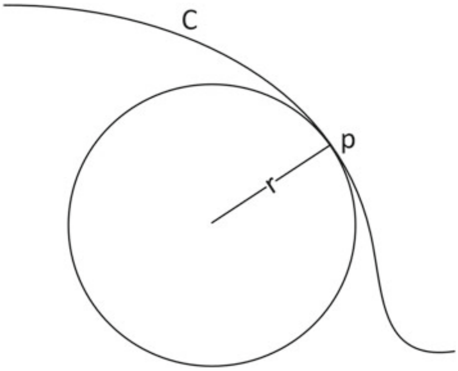


Fig. 8 Osculating circle

$$k_{Gauss} = \lim_{\epsilon \to 0} \frac{6}{\epsilon^2}(1 - \frac{C}{2\pi\epsilon}). \tag{23}$$

I.e. Gaussian curvature is a measure of how much the circumference of a circle on the manifold varies from a circle of same radius in flat space ($2\pi\epsilon$). Note that in flat space, the above equation will give value of zero for curvature. Riemannian curvature, as explained above, is defined as a deviation of vector that is parallel transported around a loop on a small parallelogram. It can also be thought of as a collection of Gauss curvatures belong to multiple sub-planes. Given two non-parallel vectors S and T, the following quantity calculated using Riemann curvature tensor is equal to Gauss curvature:

$$k_{Gauss} = R_{\mu\nu\rho\sigma} S^{\mu} T^{\nu} S^{\rho} T^{\sigma}. \tag{24}$$

Hence Riemannian curvature is equal to the Gauss curvature of the subspace times the area squared of ST parallelogram. Ricci curvature is contraction of Riemannian curvature and can be though of as average of Riemann curvature across sub-planes.

3.3 Signal Processing on Graphs

Graphs can capture spatial and topological data [7]. Examples would include computer graphics, wireless sensor networks, images, citation network analysis, computer vision (3d object correspondence), graphs can be used to represent manifolds. We do not cover characterization of neural networks through graphs in later sections, but give a overview of signal processing techniques for performing deep learning of graphs in this section

Assume G = (V, E, W) is a graph, where V represents the vertices, E the edges and W the weights assigned to the edges. We also assume undirected graphs. Then suppose

- Vertex Signal is $g(i) : V \mapsto \mathbb{R}$.
- Graph Signal is $[g(1), g(2)...g(N)]$.

Let t define the Adjacency Matrix. It is given as

$$t = D - N. \tag{25}$$

Here D is a diagonal degree matrix, which represents the number of connected edges at each vertex, and N is the weight matrix representing the strength of each edge. For a ring graph this would consist of

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

since each vertex is connected two other vertexes, and

$$N = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

since in a ring graph only adjacent vertexes are connected.

The eigenvectors of a ring graph correspond to the Fourier basis. However, for a more general graphs like the Peterson graph diagram shown in Fig. 9 we need to find the eigenvectors of t.

Let the eigenvalues be represented as $0 \geq \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_{max}$. Let the eigenvectors be represented as $\mathbf{u^{(0)}}, \mathbf{u^{(1)}}, ..., \mathbf{u^{(n-1)}}$. Then the graph Fourier transform is represented as

$$\hat{g}(\lambda_l) = \sum_{i=1}^{N} g(i)u(i)^{*l}, \tag{26}$$

and graph Inverse Fourier Transform is given by

$$g(i) = \sum_{l=0}^{N-1} \hat{g}(\lambda_l)u(i)^l. \tag{27}$$

The issues with graph Fourier transform defined above is that many operators in traditional signal processing are not directly available. Some examples include:

- Translation e.g. $f(t-3)$ is well defined in traditional signal processing. However vertices are arbitrarily assigned and translation will be ill-defined on a graph.

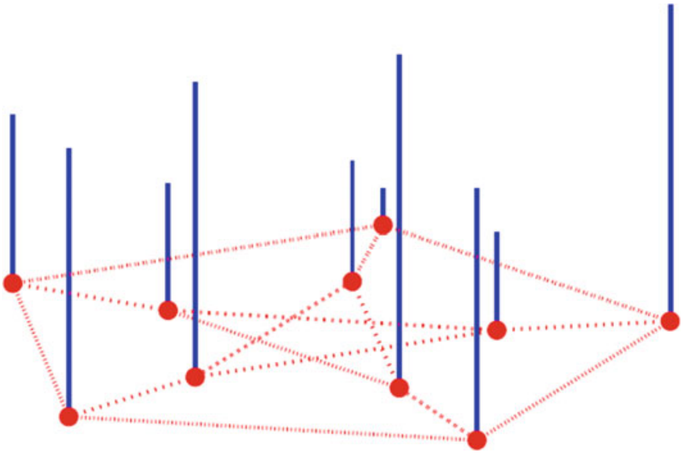


Fig. 9 Petersen graph with random positive signal value shown as the height of the bar on top of each vertex

- Modulation e.g. $e^{2\pi i\omega_0 t} f(t)$ is also well defined. However, eigenspectrum is not continuous.
- Downsampling implies reducing the samples but what does every other vertex mean in context of graph.

For example multiplication in graph spectral domain is defined as

$$\hat{g}_{out}(\lambda_l) = \hat{g}_{in}(\lambda_l)\hat{h}(\lambda_l). \tag{28}$$

The corresponding filtering can also be done in vertex domain

$$g_{out}(i) = b_{i,i} g_{in}(i) + \sum_{i,j \in N(ik)} b_{ij} g_{in}(j). \tag{29}$$

Classical convolution is defined as

$$(f * h)(t) = \int_R f(\tau)h(t-\tau)d\tau. \tag{30}$$

This cannot be directly generalized due to the term $h(t-\tau)$, which involves a translation. However, we can define

$$(g * h)(i) = \sum_{0}^{N-1} \hat{g}(\lambda_l)\hat{h}(\lambda_l)u(i)^l. \tag{31}$$

Which enforces convolution in vertex domain equals to multiplication in graph spectral domain for some constants $\{b_{i,j}\}i,\, j \in V$ Convolution can not be done in vertex domain but it an be done in spectral domain. Similar solutions can be done for modulation, dilation, coarsening and downsampling. Finally CNNs can be trained utilizing the above operator definitions and utilizing frequency domain convolutions.

4 Characterization by Homological Complexity

4.1 Betti Numbers

Let $f_N : \mathbb{R}^N \mapsto \mathbb{R}$ represent a binary classifier feed-forward neural network with N inputs and single output. Then the complexity of f_N can be thought of as the topological complexity of the set $S_N = x \in \mathbb{R}^n | f_N(x) \geq 0$. Essentially, this set represents all the inputs for which the feed-forward neural network gives a positive class.

For any subset $S \subset \mathbb{R}^n$, there exist n Betti numbers denoted by $b_i(S), 0 \leq i \leq n-1$. The first Betti number can be thought of as the number of connected components in the set, while the i-th Betti number is the number of (i + 1)-dimensional holes in S. For example, both the Sphere (S_π) and the Torus (S_τ) have first Betti number $b_0 = 1$, since both have a single connected component. The second Betti number for the sphere $b_1(S_\pi)$ is 1, but for the torus $b_1(S_\tau)$ is 2. For the sphere there is a single two dimensional hole, since only one unique (deformable) circle can be drawn on its surface. However for the torus, there are two such two-dimensional holes, the first one being the circle which can be drawn over the central hole of the torus, and the second one being the one that can be drawn across the cylindrical tube forming the torus (see Fig. 10). These two holes are not mutually deformable to the one another.

Betti numbers represent the topological notion of complexity. In particular the sum of Betti numbers of a region S_N, representing the positively classified regions of a neural network is given by $B(S_N) = \sum_i B_i(S_N)$. This can be used as a measure of complexity of classification regions of a neural network.

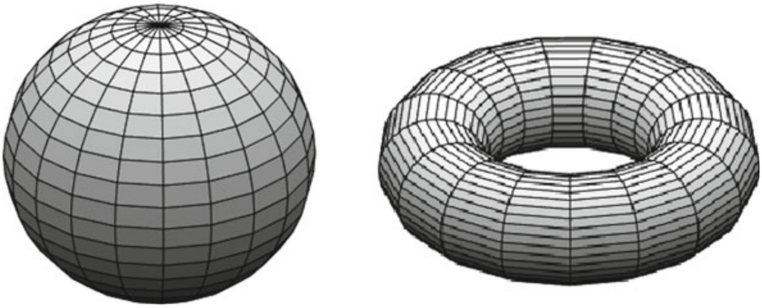


Fig. 10 Sphere (left) and Torus(right)

Table 1 Upper and lower bounds on the growth of $\mathcal{B}(S_N)$, for networks with h hidden neurons, n inputs, and l hidden layers. Architecture with many layers will be called deep, architectures with one hidden layer will be called shallow. Table taken from [5]

Inputs	Hidden layers	Activation function	Bound
Upper bounds			
n	1	Threshold	$O(h^n)$
n	1	Arctan	$O((n+h)^{n+2})$
n	1	Polynomial, degree r	$\frac{1}{2}(2+r)(1+r)^{n+1}$
1	1	Arctan	h
n	Many	Arctan	$2^{h(2h-1)}O((nl+n)^{n+2h})$
n	Many	Tanh	$2^{h(h-1)/w}O((nl+n)^{n+h})$
n	Many	Polynomial, degree r	$\frac{1}{2}(2+r^l)(1+r^l)^{(n-1)}$
Lower bounds			
n	1	Any sigmoid	$\frac{h-1}{n}^n$
n	Many	Any sigmoid	2^{l-1}
n	Many	Polynomial, deg r ≥ 2	2^{l-1}

Upper and lower bounds for sum of Betti numbers for a feed-forward neural network as a function of number of layers l, number of hidden neurons n, and number of inputs n, is derived in [5]. The table with various number of layers and activation functions is given in Table 1.

The existence of lower bound L implies that there is at-least one network N that belongs to the class and for which $B(S_N) < L$ holds, whereas the existence of an upper bound implies that U holds for all networks in the class i.e $B(S_N) > U$ for all networks.

Two important propositions that are suggested from the tables.

Proposition 4.1 *For a feed-forward neural networks with single hidden layer, the sum of Betti Numbers grows at at-most polynomial rate with the number of hidden units h, i.e.* $B(S_N) \in O(h^n)$.

Proposition 4.2 *In case there are more than one hidden units,* $B(S_N)$ *grows exponentially with number of hidden units i.e.* $B(S_N) \in \Omega(2^h)$.

The above propositions are also interlinked with the Vapnik-Chervonenkis dimension (VC-dimension) [43]. It has been proven that the VC dimension for neural networks with $arctan(\cdot)$ or $tanh(\cdot)$ is $O(p^2)$ [3]. Thus VC-dimension is polynomial with respect to number of parameters, and independent of number of layers. Since it is proven that VC-dimension is independent of number of layers per se, while the topological complexity is dependent on number of layers, it can be inferred that deeper neural networks (compared with shallow neural networks with same number of parameters) are able to tackle more complex application without losing generalization.

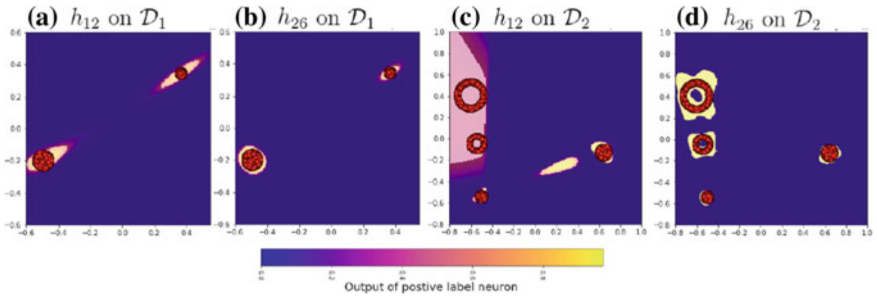


Fig. 11 The positive label outputs of single hidden layer neural networks, h_{12} and h_{26}, of 2 inputs with 12 and 26 hidden units respectively after training on datasets D_1 and D_2 with positive examples in red. Highlighted regions of the output constitute the positive decision region [17]

4.2 Architecture Selection from Homology of Dataset

The idea of measuring the homological expressiveness concerns the following problem: given a learning problem (dataset), which architecture is suitably regularized and expressive enough to learn and generalize on the given dataset. This problem can be tackled by defining a measure for complexity of dataset and characterizing neural architectures by their ability to learn subject to that complexity.

An example of two different datasets D_1 and D_2, with different homological complexities is given in figure Fig. 11. The dataset D_1 gives the positive examples sampled from the two disks and the negative samples from their complement. For dataset D_2, positive points consist of points sampled from two disk and two rings with hollow centers. We see that no single hidden layer with ≤ 12, denoted $h_{\leq 12}$, can express the two holes and clusters in D_2. On the other hand for D_1, both h_{12} and h_{26} can express decision boundary perfectly.

Definition 4.1 Given a topological space U, with Betti numbers β_n defined as the holes of dimension n, we define the homology as the sequence $H(U) = H_n(U)_{n=0}^{\infty}$ with each $H_n(U) = \mathbb{Z}^{\beta_n}$ being the n^{th} homological group. The first Betti number β^0 corresponds to the number of connected components in the topological space.

From our first example set above, $H(D_1) = \{\mathbb{Z}^1, 0, 0, 0, ...\}$ since D_1 has 2 connected components and no holes of any dimension. For the second example set $H(D_2) = \{\mathbb{Z}^4, \mathbb{Z}^2, 0, 0, 0, ...\}$, since it has 4 connected components and two holes of dimension 2.

4.3 Computational Homology

For calculating the homology of any set, we have to assume the points are sampled from an actual geometric object. Since at small scales, each data point is isolated,

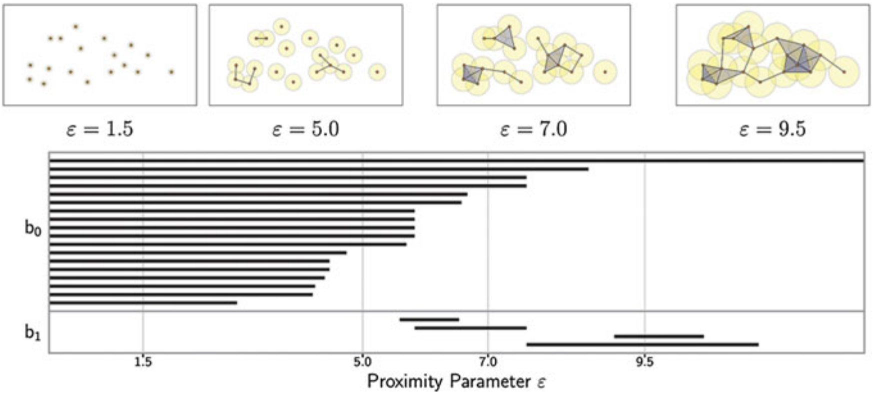


Fig. 12 Barcode from persistence homology [42]

the homology of any discrete set of points is trivially $H(D) = (Z)^M, 0, 0, 0...$ To solve this problem Zomordian and Carlsson devised persistent homology [45] based on homology of filterations of a space. Specifically, the filteration of a space X equips it with sequence of subspaces $X_0 \subset X_1 \subset X_2 ... \subset X$. A simple filteration involves growing balls of size ϵ centered on each point and letting X_ϵ be the resulting filteration. As ϵ grows, the various points merge and form connected geometric objects, leading to change in homology. There would also be new holes of various dimension which are formed. In addition to formation, holes and connected objects may vanish or merge with growing ϵ. This change in Betti numbers of X_ϵ with growing ϵ is summarized in persistence barcode diagram shown in Fig. 12. In the figure the left end point of a bar is the point at which homology detects a particular component and right end is where the component becomes indistinguishable. Suppose D is some dataset drawn from a joint distribution F, on topological space $X \times \{0, 1\}$, and X^+ denote the distribution of positive labels and X^- the distribution of negative labels. Then $H_s(f)$ denote the homology of positive decision region $f(x) > 0$. Finally let $\mathcal{F} = f : X \mapsto 0, 1$ be family of binary classifiers on X.

Theorem 4.3 *Homological Generalization. If $X = X^{-1} \cup X^{+1}$ and for all $f \in \mathcal{F}$ with $H_s(f) \neq H(X^+)$, then for all $f \in \mathcal{F}$ there exists $A \subset X^+$ so f misclassifies every $x \in A$.*

4.4 Empirical Measurements

The authors in [17] train fully connected networks with ReLU activation functions with weights of each architecture initialized to samples from normal distribution $\mathcal{N}(0, \frac{1}{\beta_0})$

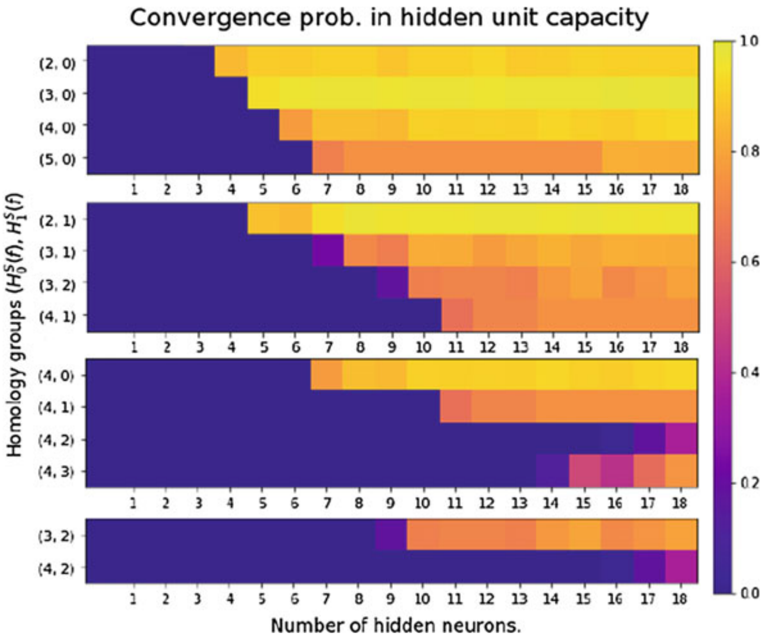


Fig. 13 Table of estimated probabilities of different neural architectures to express certain homological features of the data after training. Top: the probabilities of express homologies with increasing β_0 as a function of layers and neurons. Bottom: The probabilities of expressing $\beta_1 \in \{1, 2\}$ as a function of layers and neurons [17]

They measure the mis-classification error over course of training and the homological expressivity at the end of training, with the later term defined as

$$E_H^p(f, \mathcal{D}) = min\left\{\frac{\beta_p(f)}{\beta_p(\mathcal{D})}, 1\right\}. \tag{32}$$

The above quantity measures the capacity of model to exhibit the true homology of data. Convergence analysis of various neural networks suggest that those exhibit a statistically significant topological phase transition during learning which depends on the homological complexity of the data. For any dataset the best error of architectures with l layers and h hidden units is strictly limited in magnitude and convergence time by h_{phase}. The authors conjecture from the experimental data (see Fig. 13) that

$$h_{phase} \geq C\sqrt[l]{\beta_0 \mathcal{D}}. \tag{33}$$

Also in order to select an architecture (l, h_0) lower bounding h_{phase}, and restricting the analysis to single layer networks a lower bound estimate is obtained given by

$$\hat{h}_{phase(\beta_0,\beta_1)} \geq \beta_1 C\sqrt[l]{\beta_0(\mathcal{D})} + 2. \tag{34}$$

5 Characterization by Scattering Transform

5.1 Overview

A series of papers by Stephane Mallat [2, 28, 29] and team have utilized group theory and scattering transforms to analyze deep learning networks, assuming some simplifications. Supervised learning can be defined as method for estimating the mapping function between input data and generated labels, utilizing training data that is supplied. Let there be q samples of training data, the estimated function be $\tilde{f}(x)$ and the actual function be $f(x)$, and Ω be a subset of $\mathbb{R}^d$, which is the standard d dimensional Euclidean space. Then:

$$\{x^i, f(x^i)\}_{i \leq q} \quad \text{for} \quad x = (x(1), ...x(d)) \in \Omega. \tag{35}$$

For regression problems, $f(x)$ is in $\mathbb{R}$ and for classification $f(x)$ is class index from among all possible class indices. This problem is ill-defined if we do not make further assumptions on f. The number of points $N(\epsilon)$ we need to observe to guarantee that $|f(x) - f(x')| \leq \epsilon$, is $\approx \epsilon^{-d}$. This is an instance of curse of dimensionality implying that the number of samples required grows exponentially with dimension of data d. However if there are known regularity properties in f, we will still be able to estimate $f(x)$ without exponentially growing number of samples. Assuming that f is Lipschitz. I.e. the rate of change of output with input is bounded, is the simplest regularity assumption. Formally we require:

$$\exists C \quad \text{s.t.} \quad \forall (x, x') |f(x) - f(x')| \leq C||x - x'||. \tag{36}$$

We can possibly find a contractive operator ϕ, such that $\phi(x)$ reduces the variability of x . However the operator ϕ has to be constrained such that x belonging to different classes still are separated after contraction, i.e. $\phi(x) \neq \phi(x')$ if $f(x) \neq f(x')$. If the function is constant in certain directions, then we can carry out dimensionality reduction by projecting to a lower dimensional subspace. Otherwise, we need to linearize x through non-linear transformations $\phi(x)$ such that the $f(x)$ remains constant in certain directions. The dimensionality reduction to lower subspace can then be carried out. Effectively, we need to find $\Phi(x)$ such that $\hat{f}(x)$ is the as close estimate of $f(x)$ as possible.

$$\hat{f}(x) = p(y = 1|x) = \sigma(w^T \Phi(x) + b) \tag{37}$$

This implies that $\Phi(x)$ will linearize $f(x)$ and the following holds.

$$a^T(\Phi(x) - \Phi(x')) = 0 \implies f(x) = f(x'). \tag{38}$$

For classification this implies that we maintain the minimum distance across different classes.

$$\exists \epsilon \geq 0 \quad \forall (x, x') \in \Omega^2, |(|\phi(x) - \phi(x')|)| \geq \epsilon \quad if \quad f(x) \neq f(x') \tag{39}$$

5.2 Invariants and Symmetries

Since the input data lie on highly curved manifolds, f will linearize input successively through the layers. Since classification maps regions in the input manifolds to specific indices, these regions can be considered as level sets defined as $\Omega_t = x : f(x) = t$, where f is continuous. Since at the final layer we are able to fully linearize the input and map different classes to different hyperplanes, the learned w is such that $f(x)$ is approximated by $< \phi(x), w >$. Also if x belongs to class t, then $< \phi(x), w > \approx t$. Global symmetry can be thought of as an operator φ that leaves f invariant:

$$\forall x \in \Omega, \ f(\varphi(x)) = f(x) \tag{40}$$

The impact of the symmetries can be of two distinct but related forms:

(i) Invariance: $\phi(\varphi(x)) = \varphi(x)$ for each x
(ii) Equivariance: $\phi(\varphi(x)) = \varphi(\phi(x))$ for each x, φ

To satisfy the above, we need invertible operators g such they leave the value of f unchanged i.e. $f(g.x) = f(x)$ for all $x \in \Omega$. We note that composition of two global symmetries g_1 and g_2 is also a global symmetry $g_1.g_2$. Since an identity and inverse element is always present, and group closure holds, these symmetries form a group. Such differentiable manifolds with a group structure are called Lie groups [14]. We say that G is group of local symmetries of f if:

$$\forall x \in \Omega, \exists C_x > 0, \quad \forall g \in G \quad with \quad |g|_G < C_x, \quad f(g.x) = f(x) \tag{41}$$

Examples of symmetries for images include

(i) Translations $\{\varphi_v, ; v \in \mathbb{R}^2\}$, with $\varphi_v(x)(u) = x(u - v)$.
(ii) Dilations $\{\varphi_s; s \in \mathbb{R}_+\}$, with $\varphi_s(x)(u) = s^{-1}x(s^{-1}u)$.
(iii) Rotations $\{\varphi_\theta; \theta \in [0, 2\pi]\}$, with $\varphi_\theta(x)(u) = x(R_\theta u)$.
(iv) Mirror Symmetry: $\{e, M\}$, with $Mx(u_1, u_2) = x(-u_1, u_2)$.

All the above transformations can be combined in the affine group $Aff(\mathbb{R}^2)$ with 6 degrees of freedom in the representation.

5.3 Translation and Diffeomorphisms

Global symmetries are tough to find so we first use local symmetries. These can be modeled as a group of translations and diffeomorphisms which deform signals locally. In image and speech recognition applications, no high dimensional symmetry groups exist, however stability to local deformations is expected (see Fig. 14). Let

$$x \in L^2(\mathbb{R}^m), \tau : \mathbb{R}^m \mapsto \mathbb{R}^m. \tag{42}$$

Then

$$x_\tau = \varphi_\tau(x), x_\tau(u) = x(u - \tau(u)). \tag{43}$$

where φ_τ is a change of variables. The term x_τ is deforming pixel locations, instead of pixels themselves.

The simplest Lie Group of transformations is the translation group $G = \mathbb{R}^n$. The action of $g \in G = R^n$ over $x \in \Omega$ is $g.x = x(u - g)$. Since translations are defined by limited number of parameters (only two in images, offsets in x and y dimensions), those are not very powerful symmetries. Also diffeomorphism symmetries are application and dataset specific e.g. in case of MNIST digits, some transformation leave the digit unchanged while other would change it [25]. For linearizing local symmetries we use the transformation $\phi(x)$ which linearizes the action of $g \in G$ locally. By definition, Φ is Lipschitz continuous if

$$\exists C > \forall(x, g) \in \Omega \times G, ||\Phi(g.x) - \Phi(x)|| \geq C|g|_G||x|| \tag{44}$$

Here $|g|_G$ measures difference between $g \in G$ and identity and is the euclidean norm of $g \in \mathbb{R}^n$. We note that a bounded operator g closely approximates $\phi(x) - \phi(g.x)$ if the norm of g is sufficiently small.

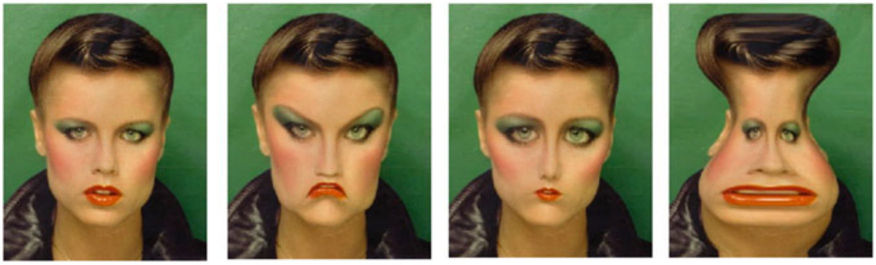


Fig. 14 Local deformations in an image [8]

5.4 Contraction and Scale Separation by Wavelets

To evaluate stability, we first need to quantify the amount of deformation. Also, we need to find a notion of scale: in many applications, we are interested in local invariance rather than global group invariance. Deep convolutional networks are covariant rather than invariant to translations. Hence translation in the input lead to translations in the output for convolutional layers. However to make the computation invariant to translations scales need to be separated and non-linearities need to be applied. A linear operator computes local invariants to action of translation group G. This is done by taking mean of x on the orbit $\{g.x\}_{g \in G}$. This is done by convolution with a kernel $\phi_J(u) = 2^{-nJ}\phi(2^J u)$ of size 2^J with $\int \phi(u)du = 1$:

$$\phi_J x(u) = x \star \phi_J(u). \tag{45}$$

This is verifiable that the averaging is Lipschitz continuous to diffeomorphisms for all $x \in L^2(R^n)$ over a translation range 2^J. However, it eliminates variations in the input above frequency 2^J. If $J = \infty$ it eliminates almost all information.

We use wavelets of different scale to separate variations at different scales. K wavelets are defined $\psi_k(u)$ of $u \in R^n$. These are dilated by $2^j : \psi_{j,k}(u) = 2^{-jn}\psi_k(2^{-j}u)$. Using convolutions with wavelets we can compute the average of x at scales 2^J and variations at scales $2^j \geq 2^J$. This convolution operation is contractive and the representation obtained is sparse. For audio signals n = 1, and K = 12 intermediate frequencies are used within octave 2^J. For images n = 2, we utilize $\frac{\pi k}{K}$ orientations which are rotated. e.g. we use J = 4, K = 4, as shown in Fig. 15.

5.5 Filter Bank, Phase Removal and Contractions

We utilize a cascade of filters at different scales to compute the scattering transform with wavelets. At each scale we use filters $w_{j,k}$ which compute the wavelets $\psi_{j,k} = w_{j,k} * \phi_{j-1}$. Also at every stage we perform averaging at a increased scale by utilizing the filter $w_{j,0}$ to compute $\phi_j = w_{j,0} * \phi_{j-1}$. Wavelet coefficients $x_j(u,k) = x * \psi_{j,k}(u)$ oscillate at scale 2^j, and averaging x_j with ϕ_j would output a zero signal. Hence non-linearities are required to remove oscillations. Modulus($\rho(\alpha) = |\alpha|$) is one such non-linearity which computes the positive envelope. We can also use the ReLU given by max(m,0), which is also contractive operator similar to modulus. Any non-linear operator for which $|\rho(\alpha) - \rho(\alpha')| \leq |\alpha - \alpha'|$ can be considered contractive. Averaging $(\rho(x * \psi_{j,k}(u))$ with ϕ_J output positive coefficients which are locally invariant at scale 2^J. Local multiscale invariant examples are mel-spectrum in speech and SIFT in images. They have information loss due to averaging and hence are calculated at small scales (e.g. 16^2 pixels for SIFT). Due to this they don't capture large scale structure and also fail to capture scale interactions. Scattering transform using wavelet modulus operators is shown in Fig. 16.

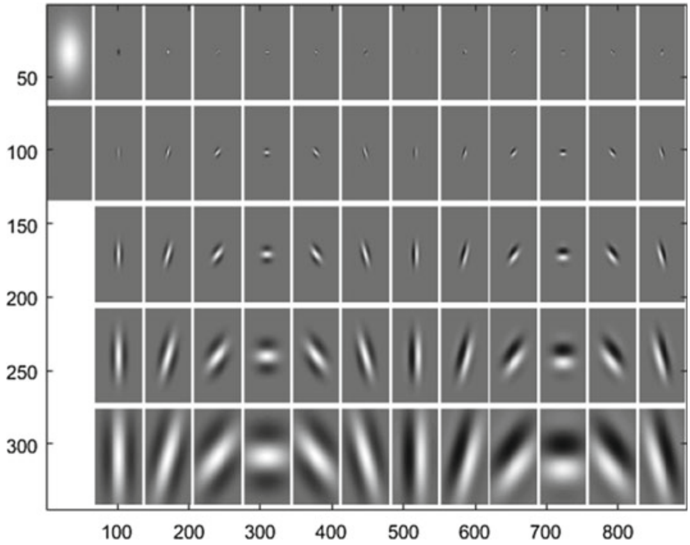


Fig. 15 Wavelet filters [29]

5.6 Translation Groups

Deep neural networks are composed of a cascade of linear filters (convolution or fully connected) layers interleaved with point wise non-linearities ρ. For simplicity, an architecture where the convolutional filter do not add across the input channels is used.

Suppose $x_j(u, k_j)$ is computed by convolving single channel $x_{j-1}(u, k_{j-1})$ along u, where j is the layer index. Then

$$x_j(u, k_j) = \rho(x_{j-1}(., k_{j-1}) * w_{j,h}(u)) \quad \text{with} \quad k_j = (k_{j-1}, h). \tag{46}$$

Iterating over j defines a convolution tree

$$x_J(u, k_J) = \rho(\rho(\rho(\rho(x * w_{1,h_1}) * ...)*) * w_{J-1,h_{J-1}})) * w_{J,h_J})). \tag{47}$$

For m = 1, coefficients $x_J(u, k_J) = \rho(x * \psi_{j_1,k_1}) * \phi_J(u)$ are the wavelet coefficients. For m = 2, $\rho(\rho(x * \psi_{j_1,k_1}) * \psi_{j_2,k_2}) * \phi_J(u)$ are complementary invariants measuring interactions of x at scale 2^{j_1} within a distance 2^{j_2} and along orientation and frequency bands defined by k_1 and k_2. See Fig. 17. It is noted that in case of images and speech most of the energy is contained in the first two stages i.e. $m \leq 2$.

If x is stationary than $\rho(...\rho((x * \phi_{j_1,k_1}) * \psi_{j_2,k_2}...)$ remains stationary because convolutions and point-wise operators preserve stationarity. For a rectifier or modulus $\rho(\alpha) = \alpha$ for $\alpha \geq 0$. So the ρ at output of averaging filter can be removed. For a band-

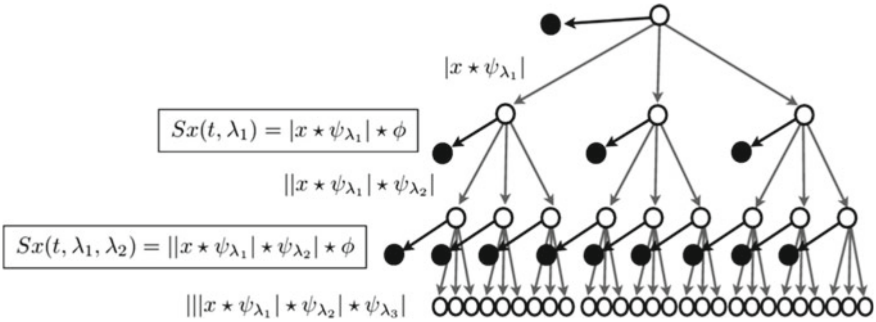

$|x \star \psi_{\lambda_1}|$
$Sx(t, \lambda_1) = |x \star \psi_{\lambda_1}| \star \phi$
$||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}|$
$Sx(t, \lambda_1, \lambda_2) = ||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}| \star \phi$
$|||x \star \psi_{\lambda_1}| \star \psi_{\lambda_2}| \star \psi_{\lambda_3}|$

Fig. 16 The scattering transform includes wavelet operators followed by modulus operators in cascade forming a tree structure. At each stage, the coefficients are also averaged through the low pass operator ϕ [2]

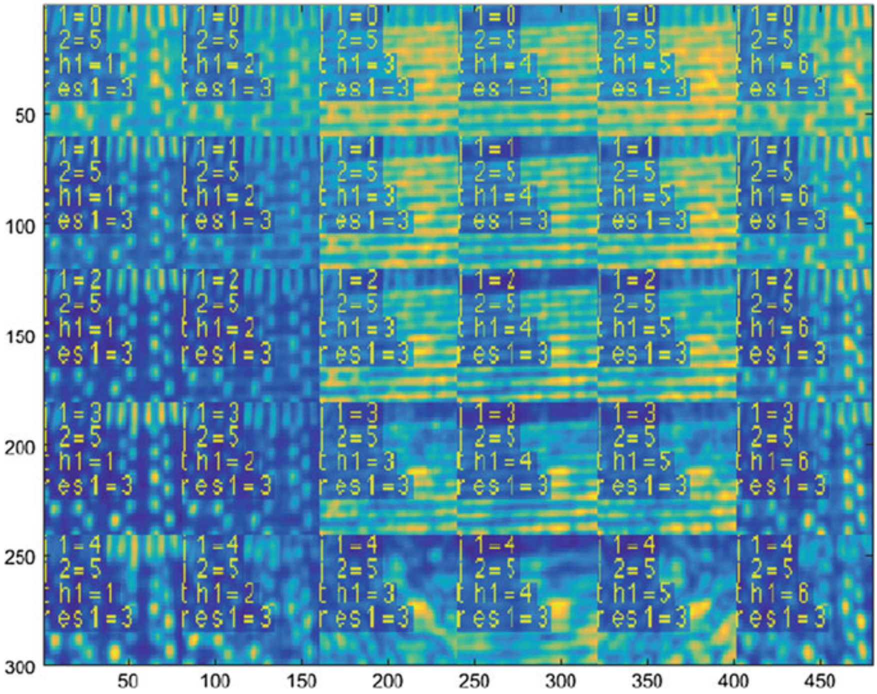


Fig. 17 Scattering coefficients

pass filter the non-linearity removes the phase or sign, which has strong contraction effect. We can remove ρ from all low pass filters, then cascade of J convolutions reduces to m (number of bandpass filters).

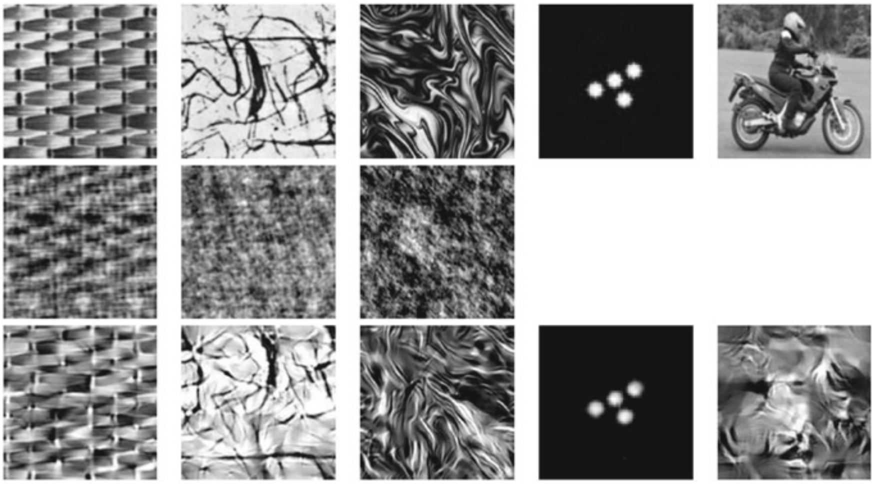


Fig. 18 Inverse scattering compared to Gaussian process. First row: original textures. Second row: Gaussian process with covariance identical to first row. Third row: inverse scattering performed with first two orders of scattering coefficients [29]

5.7 Inverse Scattering and Sparsity

Scattering transforms are not invertible. However for a given $\phi_J(x)$ we can try to find $\tilde{x}$ such that $||\phi_J(x) - \phi_J(\tilde{x})|| < \sigma_J$. This can be possibly be achieved by initializing $\tilde{x}_0$ with white Gaussian noise and using gradient descent while trying to reduce $||\phi_J(x) - \phi_J(\tilde{x_n})||$. In the Fig. 18, first row is the original texture. The second row shows a Gaussian process with same covariance as the original texture in first row. The third row shows the inverse scattering as described above using scattering coefficients with order $m <= 2, 2^J = N$ and $K = 8$. As can be seen from the Fig. 18, scattering coefficients are able to reproduce much more likeness in texture than gaussian process with first two moments identical to original figure.

6 Characterization by Curvature

6.1 Mean Field Theory and Gaussian Curvature

The method described in [34] analyze deep neural networks using a combination of Riemannian curvature and mean field theory. For simplicity of analysis they assume that neural networks are having random weights. They driving intuition here is that DNNs can represent generic random functions. Also DNN can disentangle curved manifolds at the input into flat manifolds at output. Finally deeper neural networks with same number of neurons can do this disentanglement more effectively than shallow networks.

Consider a DNN with D layers of weights $w^1 ... w^D$ and $D + 1$ layers of vectors $x^0, ... x^D$ with N_l neurons in layer l, $x^l \in \mathbb{R}_l^N$ and $w^l \in \mathbb{R}^{N_l \times N_{l-1}}$. The output of filter w and biases b^l, with non-linearity ϕ at layer l, for an input x^0 are given is given by:

$$x^l = \phi(h^l), \quad h^l = w^l x^{l-1} + b^l \tag{48}$$

Here h^l is the out of the filter at layer l, and ϕ is a point-wise non-linearity that acts on h^l to generate x^l. The weights and biases are initialized with gaussian random variables distributed as follows:

- $w_{ij}^l \sim \mathbf{N}(0, \sigma_\omega^2 / N_{l-1})$.
- $b \sim \mathbf{N}(0, \sigma_b^2)$.

As a simple manifold propagates forward through the neural network, the modification in its geometry is to be measured. The normalized squared length at input of layer l is defined as

$$q^l = \frac{1}{N_l} \sum_{l-1}^{N_l} (h_i^l)^2 \tag{49}$$

Through central limit theorem, this quantity will converge to zero mean Gaussian random variable for large N_l. As this distribution is processed through the layers and iterative map of q^l based on layer index l can be obtained in Eq. 49

$$q^l = V(q^{l-1} | \sigma_\omega, \sigma_b) = \sigma_\omega^2 \int Dz \phi(\sqrt{q^{l-1}} z) + \sigma_b^2 \quad \text{for} \quad l = 2, ..., D, \tag{50}$$

where

$$Dz = \frac{dz}{\sqrt{2\pi}} e^{-z^2/2} \tag{51}$$

is the standard Gaussian measure and z is a dummy variable and initial condition is

$$q^1 = \sigma_w^2 q^0 + \sigma_b^2 \tag{52}$$

and q^0 represents the length in to the first layer defined as

$$q^0 = \frac{1}{N_0} x^0 x^0. \tag{53}$$

The function V (in Eq. 50) is iterative variance, which predicts change of length as input passes through the network. It is a concave function whose interaction with unity line determines its fixed points $q^*(\sigma_\omega, \sigma_b)$. As can be seen from Fig. 19

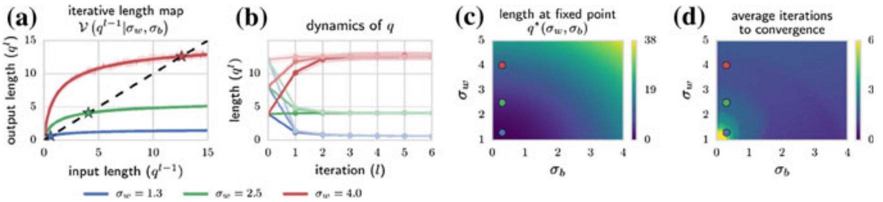


Fig. 19 Variation of squared q^l for network with tanh non-linearity and 1000 hidden units. **a** And length map has been shown for different σ_w at σ_b. Solid lines represent theoretical predictions while simulation results are shown with dots. Fixed points q^* of the map are shown as stars. **b** The length map changes through the layers and converges within a few layers to q^* in all cases (lines=theory; dots=simulation). **c** The fixed point is shown as a function of σ_w and σ_b. **d** Number of layers in which the fractional deviation from the fixed point is less than one. The $(\sigma_b; \sigma_w)$ pairs in (**a, b**) are marked with color matched circles in (**c, d**) [34]

1. for $\sigma_b = 0$, $\sigma_\omega < 1$, the only intersection is $q^* = 0$, and hence network shrinks all inputs to zero.
2. for $\sigma_w > 1$ and $\sigma_b = 0$, $q^* = 0$, the fixed point becomes unstable and length map acquires second nonzero but stable fixed point, and the network contracts large inputs and expands small inputs.
3. for any nonzero bias σ_b the length map has single stable non-zero fixed point. In this case the injected bias prevents the signal from decaying to zero.

Transient Chaos: If we consider two inputs, $x^{0,1}$ and $x^{0,2}$, the geometry of these two input can be represented by 2×2 matrix for inner products given by.

$$q^l_{ab} = \frac{1}{N} \sum_{i=1}^{N_l} h^l_i(x^{0,a}) h^l_i(x^{0,b}) \quad a, b \in 1, 2. \tag{54}$$

The terms q^1_{11} and q^l_{22} is the length that can be directly inferred from Eq. 50. For the non-diagonal terms, a correlation map C can be inferred for q^l_{12}:

$$\begin{aligned} q^l_{12} &= C(c^{l-1}_{12}, q^{l-1}_{11}, q^{l-1}_{22} | \sigma_\omega, \sigma_b) \\ &= \sigma^2_\omega \int Dz_1 Dz_2 \phi(u_1)\phi(u_2) + \sigma^2_b, \end{aligned}$$

where

$$u_1 = \sqrt{q^{l-1}_{11}} z_1, \tag{55}$$

$$u_2 = \sqrt{q^{l-1}_{22}} \left[c^{l-1}_{12} z_1 + \sqrt{1 - (c^{l-1}_{12})^2 z_2} \right], \tag{56}$$

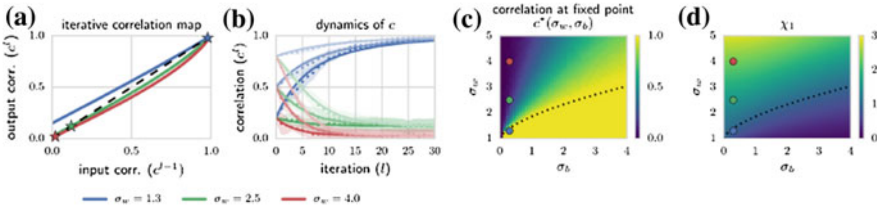


Fig. 20 Variation of the correlation, c^l_{12}, as it passes through layers in randomly weighted network with tanh non-linearity. **a** The C-map is shown for the case σ_w and σ_b = 0.3 as in Fig. 19. **b** The change in C-map is shown. Solid lines are theory whereas dots are for simulations using N_l equal to 1000. **c** Fixed points c^* of the C-map. **d** The slope of the C-map at 1, χ_1, partitions the space (black dotted line at $\chi_1 = 1$) into chaotic ($\chi_1 > 1, c^* < 1$) and ordered ($\chi_1 < 1, c^* = 1$) regions [34]

and correlation coefficient is defined as:

$$c^l_{12} = q^l_{12}(q^l_{11}q^l_{22})^{-1/2}. \tag{57}$$

Here z_1 and z_2 are uncorrelated Gaussian variables, while u_1 and u_2 are correlated Gaussian variables with covariance matrix $< u_a u_b >= q^{l-1}_{ab}$.

Since length of each point converges rapidly to $q^*(\sigma_\omega, \sigma_b)$ as we propagate through the network we can substitute this value. We can compute c^* by setting $q^l_{11} = q^l_{22} = q^*(\sigma_\omega, \sigma_b)$ and dividing by q^*. We can thus obtain an C-map (correlation coefficient map):

$$c^l_{12} = \frac{1}{q^*} C(c^{l-1}_{12}, q^*, q^* | \sigma_\omega, \sigma_b). \tag{58}$$

The C-map always has a fixed point $c^* = 1$. The stability of fixed point depends on the slope of the map at 1, which is

$$\chi_1 = \frac{\partial c^l_{12}}{\partial c^{l-1}_{12}}|_{c=1} = \sigma^2_\omega \int Dz[\phi'(\sqrt{q^*}z)]^2.$$

Then three regions are possible based on value of χ_1 (see Fig. 20):

- for small σ_ω, $c^* = 1$ is the only fixed point and is stable as $\chi_1 < 1$ and any two points converge as they propagate through network.
- as σ_ω increases, χ_1 crosses 1 and a new c^* is created, different from 1.
- for larger σ_ω the strong weights overwhelm the biases making the input decorrelate and orthogonal, leading to stable fixed point at $c^* = 0$.

Hence $\chi_1(\sigma_w, \sigma_b) = 1$ yields a phase transition boundary in the (σ_w, σ_b) plane separating it into chaotic phase and ordered phase based upon separation/convergence. We note that log χ_1 is the Lyapunov exponent in dynamical system theory.

Propagation Through Deep Net. We can now try to track the length of a manifold as it propagates through a deep neural network. At the first layer

$$h^l(0) = h^l(x^0(\theta)), \tag{59}$$

where $x^0(\theta)$ is a 1-D manifold with θ being intrinsic scaler coordinates. We can choose a circle defined as:

$$\sqrt{N_1 q}\left[u^0 cos(\theta) + u^1 sin(\theta)\right], \tag{60}$$

where u^0 and u^1 are orthonormal basis for 2-dimensional subspace of R^N. At each point, the manifold $h(\theta)$ has tangent or velocity vector.

$$v(\theta) = \partial_\theta(h(\theta)). \tag{61}$$

Curvature is related to acceleration which is defined as rate of change of velocity.

$$a(\theta) = \partial_\theta(v(\theta)) \tag{62}$$

At each point θ, $v(\theta)$ and $a(\theta)$ span a 2D space of R^N. The osculating circle is defined as circle with radius $R(\theta)$ such that it has same position, velocity and acceleration as $h(\theta)$ at θ

Extrinsic curvature is defined as

$$k(\theta) = \frac{1}{R(\theta)}, \tag{63}$$

Also extrinsic curvature is related to velocity and acceleration as:

$$k(\theta) = v.v - 3/2\sqrt{(v.v)(a.a) - (v.a)^2}. \tag{64}$$

Total curve length as per Euclidean metric is

$$\mathbf{L}^E = \int \sqrt{g^E(\theta)} d\theta. \tag{65}$$

where

$$g^E(\theta) = v(\theta).v(\theta). \tag{66}$$

For k dimensional manifold $\mathcal{M}$ embedded in $\mathbb{R}^N$ the Gauss map maps a point $\theta \in M$ to its k dimensional tangent plan $T_\theta M \in \mathbf{B}_{k,n}$ where $\mathbf{B}_{k,n}$ is Grassmanian manifold of all k-dimensional spaces. Gauss Map takes a point θ on the curve and maps it to unit velocity vector $\hat{v}(\theta) = v(\theta)/\sqrt{v(\theta).v(\theta)}$. Gauss metric is related to extrinsic curvature and Euclidean metrics by

$$g^G(\theta) = k(\theta)^2 g^E(\theta). \tag{67}$$

Length of curve under Gauss Map is

$$L^G = \int \sqrt{g^G(\theta)} d\theta. \tag{68}$$

Extrinsic curvature and Euclidean metric change iteratively as they propagate through the layers as

$$\begin{aligned}
g^{E,l} &= \chi_1 g^{E,l-1}, \\
(k^l)^2 &= 3\frac{\chi_2}{\chi_1} + \frac{1}{\chi_1}(k^{l-1})^2, \\
g^{E,l} &= q^*, \\
(k^1)^2 &= 1/q^*, \\
\chi_2 &= (\sigma_\omega)^2 \int Dz[\phi''(\sqrt{q}z]^2,
\end{aligned}$$

where χ_2 is defined analogously to χ_1.

If the circle is scaled up i.e. $h_\theta = \chi h(\theta)$, then the length of L_E and radius scale up by χ but curvatures scales by χ^-1 so L^G doesn't change. However, in deep networks this is not the case and L^G also increases. In sigmoidal neural networks the evolution behaves differently depending on value of χ_1.

For Chaotic phase $\chi_1 > 1$: The euclidean metric g^E grows exponentially with depth due to multiplicative stretching through χ_1. The stretching attenuates any curvature in layer $l - 1$ by factor of $1/\chi_1$ but new curvature is added due to χ_2, due to single neuron non-linearity. So unlike linear expansion, extrinsic curvature is not lost but ultimately approaches k^*. This implies global curvature measure L^G grows exponentially with depth (see Fig. 21).

This shows that DNNs become highly curved functions allowing to compute exponentially convex function over simple low dimensional manifolds. Also the length of curve grows exponentially with depth but only grows as square root with width.

6.2 Riemannian and Ricci Curvature Measurement

The previous technique relies on analysis of neural network with random weights. However, the real-world networks are trained using stochastic gradient descent and hence are not truly random. Also, the above technique does not give a fine-grained view of behavior the neural network for various multi-dimensional manifolds of transformations. A recent algorithm [20] that works on premise of analyzing trained

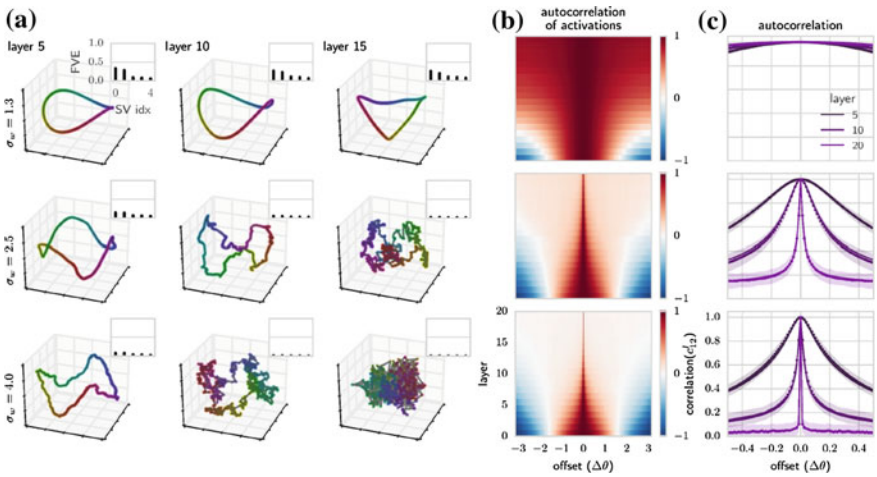


Fig. 21 A one dimensional manifold circle is being input to three different neural networks with different σ_w and fixed $\sigma_b = 0.3$. **a** PCA is used for projecting the hidden layer outputs in simulations to a 3 dimensional subspace for plotting. Only layer 5, 10 and 15 are shown. The inset graph shows relative energy of first five principal components. For $\sigma_w = 4$, the maximum energy singular values are more evenly distributed, and the circle gets more tangled with each succeeding layer. **b** The variation in autocorrelation, $c_{12}^l(\Delta\theta) = \int d\theta q^l(\theta; \theta + \Delta\theta)/q^*$, is shown across multiple layers. **c** The theoretical prediction for the auto-correlation along with mean and standard deviation of measured auto-correlation is shown. From [34]

neural networks based on differential geometric context, tries to overcome these shortcomings. It starts with the following ideas and assumptions.

1. the datasets used for various machine learning problems constitute low dimensional manifolds embedded in higher dimensional spaces.
2. the classification problem entails segregation of disjoint non-linear manifolds inside the dataset.
3. these classes in the input lie on manifolds that are not linear, and hence linear methods cannot be used to segregate them.
4. deep learning networks transform the input non-linear manifold, through various transformations it learns during training to a linear region. In various deep learning architectures, these transformation are implemented through a combination of linear or affine layers in conjunction with non-linear activations. We can consider these combined transformations a change of basis over various highly curved coordinate systems.
5. once the non-linearized input is converted to linear region through these cascaded transforms, we can use simple linear projection technique of type $Wx + b$ (where W is the linear matrix of weights, x the input vector, and b the bias term) which will effectively partition the spaces through hyperplanes.

6. since the transforms are over curved basis, we can use the tools from General Theory of Relativity which uses the mathematics of Riemannian curvature extensively.
7. we measure the curvature by calculating the gradient of the output compared to a set of parameterized transform in input data. I.e. $\frac{\partial o}{\partial t}$, here o is the output and t is the transform parameter.

We attempt to perform a direct measurement of Riemannian curvature for a manifold of transformations, which can have one or more dimensions. The transforms selected depend on the type of data, and for images we can use transforms like translation or rotation or a combination of these. As we move the input validation set over the manifold of transformations, we need to measure the gradient of output using difference equations. The steps for these measurements are shown in Fig. 22. The various steps in the are described previously in Sect. 3.2.

One Dimensional Manifold Classifier. We test the above algorithm on a synthetic dataset and perform the calculations with transformations consisting of a single dimensional manifold. We feed this single dimensional manifold of transformed inputs to a pre-trained MLP trained to discriminate between two archimedian spirals. We use two archimedean spirals separated by π radians, since those are not linearly separable. The spiral are as given below:

$$x_i(\theta) = t\,cos(\theta + d_i) + n(\theta), \tag{69}$$

$$y_i(\theta) = t\,sin(\theta + d_i) + n(\theta), \tag{70}$$

where the two equations generate the two rectangular coordinates for all values of angle θ between 0 to 2π, while t is a constant. The variable d_i is the initial angle of the spiral which is selected to be 0 for first spiral and or π for the second spiral. In the training set, we also add i.i.d Gaussian noise to the co-ordinates generated in form of $n(\theta)$.

For generating the dataset for the neural network, we need to create a set of vectors from the coordinates generated above. For this we utilize two Gaussian random vectors with unity variance, u_0 and u_1, each of length 1000, which we multiply with the previously generated coordinates to create the input dataset for the neural network.

$$v = x_i(\theta)u_0 + y_i(\theta)u_1. \tag{71}$$

For creating the training set, we generate random θ for each training sample to be generated, and calculate v using this θ as show in the equation (71) above. This v is the input training vector for training the neural network. A particular instance of training set is represented in Fig. 23, where red dots correspond to the first class and blue dots to the second class. For the validation dataset (which will be used to measure curvature), we do not generate x and y through the spiral equations, but

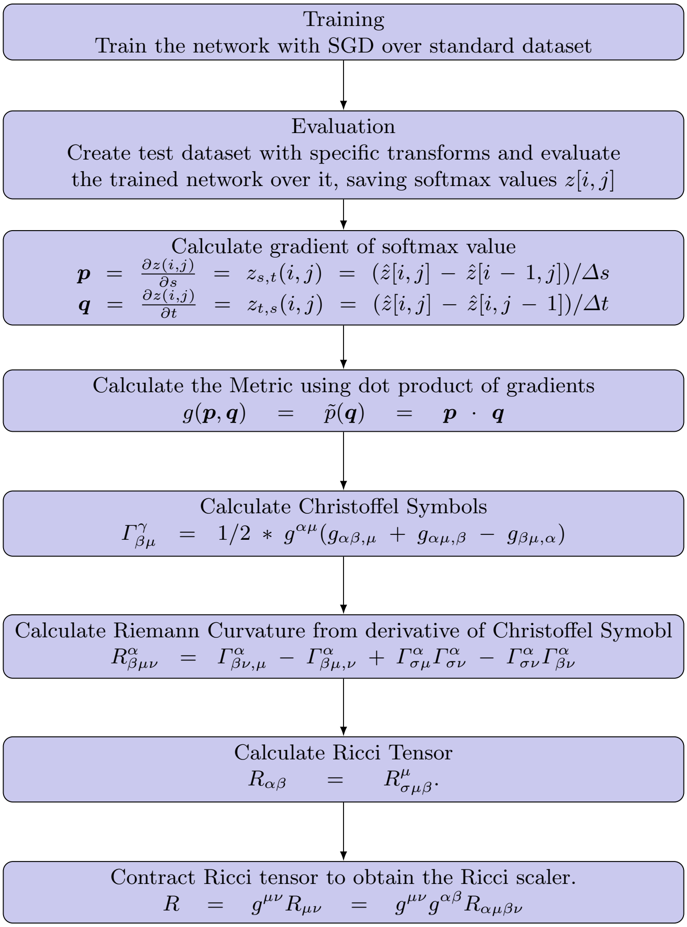

Training
Train the network with SGD over standard dataset
Evaluation
Create test dataset with specific transforms and evaluate the trained network over it, saving softmax values $z[i,j]$
Calculate gradient of softmax value
$\boldsymbol{p} = \frac{\partial z(i,j)}{\partial s} = z_{s,t}(i,j) = (\hat{z}[i,j] - \hat{z}[i-1,j])/\Delta s$
$\boldsymbol{q} = \frac{\partial z(i,j)}{\partial t} = z_{t,s}(i,j) = (\hat{z}[i,j] - \hat{z}[i,j-1])/\Delta t$
Calculate the Metric using dot product of gradients
$g(\boldsymbol{p},\boldsymbol{q}) = \tilde{p}(\boldsymbol{q}) = \boldsymbol{p} \cdot \boldsymbol{q}$
Calculate Christoffel Symbols
$\Gamma^{\gamma}_{\beta\mu} = 1/2 * g^{\alpha\mu}(g_{\alpha\beta,\mu} + g_{\alpha\mu,\beta} - g_{\beta\mu,\alpha})$
Calculate Riemann Curvature from derivative of Christoffel Symobl
$R^{\alpha}_{\beta\mu\nu} = \Gamma^{\alpha}_{\beta\nu,\mu} - \Gamma^{\alpha}_{\beta\mu,\nu} + \Gamma^{\alpha}_{\sigma\mu}\Gamma^{\alpha}_{\sigma\nu} - \Gamma^{\alpha}_{\sigma\nu}\Gamma^{\alpha}_{\beta\nu}$
Calculate Ricci Tensor
$R_{\alpha\beta} = R^{\mu}_{\sigma\mu\beta}.$
Contract Ricci tensor to obtain the Ricci scaler.
$R = g^{\mu\nu}R_{\mu\nu} = g^{\mu\nu}g^{\alpha\beta}R_{\alpha\mu\beta\nu}$

Fig. 22 Steps in calculating Ricci scaler curvature

instead generate entire gird of possible x_i and y_i. We then utilize Eq. 71 to generate the validation vectors.

Once the datasets is created and network is trained, we test the network against the validation set and store the output softmax values. This softmax output is used to find various gradients and curvature metrics as shown in Fig. 22. We firstly note that the two classes are on single dimensional manifold (of parameter θ), but are not linearly separable. From the plotted classifier regions as shown in Fig. 24, it can be inferred that the network correctly discriminates between the two classes. The only exception is at the center, where we have some overlap. We plot the Ricci scaler values for the entire validation set as shown in Fig. 25. We note that the networks learns very large curvature values for the boundary regions separating the two classes. We infer that

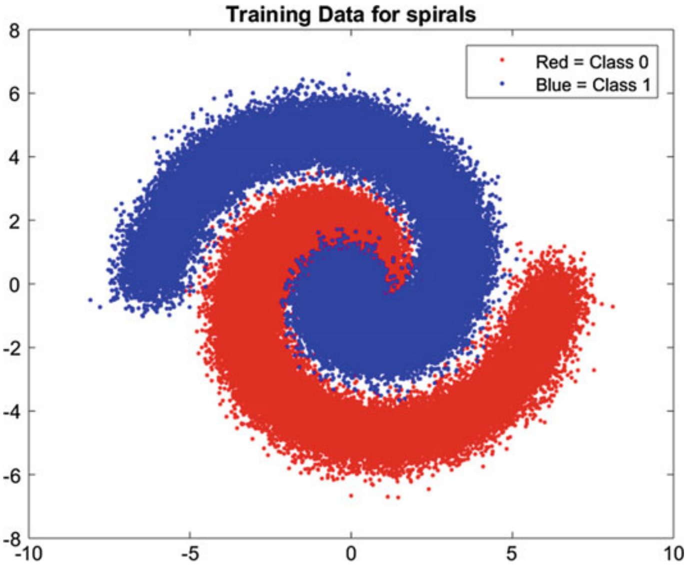


Fig. 23 Training set with red for class 0, and blue for class 1

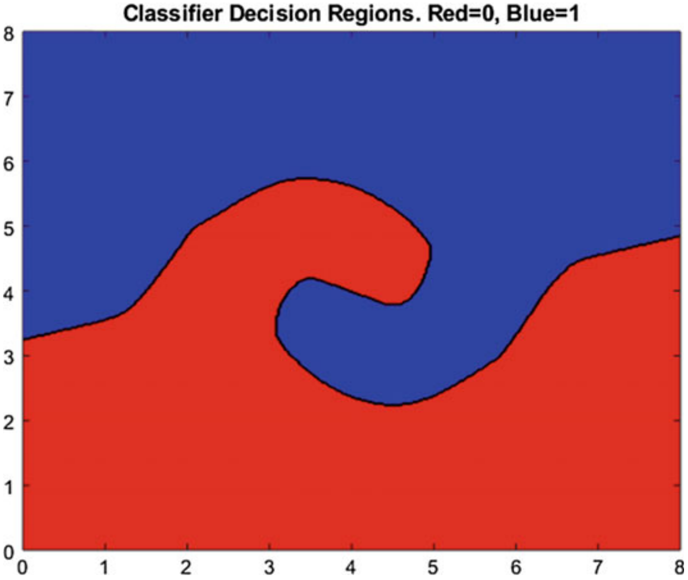


Fig. 24 Classifier regions for trained neural net

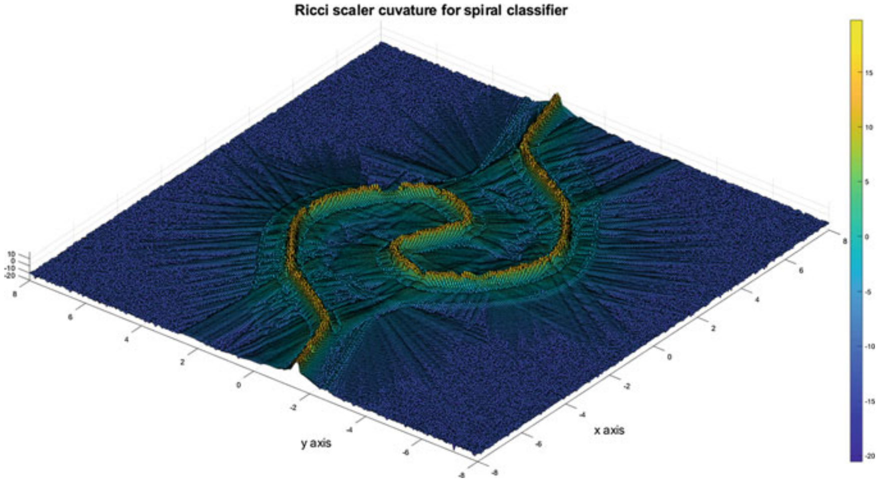


Fig. 25 Ricci scaler for spiral data

the neural network is increasing the distance between classes at boundary regions to be able to discriminate between the classes.

We can utilize the above observation to note that any boundary regions with small curvature may be possible areas where small deformations/transformations in input data may lead to classification errors. Another interesting observation is that the neural networks learns to classify in certain predictable ways even in the regions where we haven't supplied any training data.

References

1. Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Cheng, Q., Chen, G., et al. Deep speech 2: end-to-end speech recognition in English and mandarin. In: International Conference on Machine Learning, pp. 173–182 (2016)
2. Andén, J., Mallat, S.: Deep scattering spectrum. IEEE Trans. Signal Process. **62**(16), 4114–4128 (2014)
3. Bartlett, P.L., Maass, W.: Vapnik-Chervonenkis dimension of neural nets. In: The Handbook of Brain Theory and Neural Networks, pp. 1188–1192 (2003)
4. Bengio, Y., Delalleau, O.: On the expressive power of deep architectures. In: International Conference on Algorithmic Learning Theory, pp. 18–36. Springer, Berlin (2011)
5. Bianchini, M., Scarselli, F.: On the complexity of shallow and deep neural network classifiers. In: ESANN (2014)
6. Bredon, G.E.: Topology and Geometry, vol. 139. Springer Science & Business Media (2013)
7. Bronstein, M.M., Bruna, J., LeCun, Y., Szlam, A., Vandergheynst, P.: Geometric deep learning: going beyond Euclidean data. IEEE Signal Process. Mag. **34**(4), 18–42 (2017)
8. Bruna, J.: Geometric stability in Euclidean domains: the scattering transform and beyond. https://joanbruna.github.io/MathsDL-spring18/ (2018)
9. Bruna, J., Mallat, S.: Invariant scattering convolution networks. IEEE Trans. Pattern Anal. Mach. Intell. **35**(8), 1872–1886 (2013)

10. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation (2014). arXiv preprint arXiv:1406.1078
11. Choquet-Bruhat, Cécile, Y., DeWitt-Morette, C., Dillard-Bleick, M.: Analysis, Manifolds, and Physics. Gulf Professional Publishing (1982)
12. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley (2012)
13. Friedman, J., Hastie, T. and Tibshirani, R.: The Elements of Statistical Learning, vol. 1. Springer Series in Statistics. Springer, New York (2001)
14. Gilmore, R.: Lie Groups, Lie Algebras, and Some of Their Applications. Courier Corporation (2012)
15. Glorot, X., Bordes, A., Bengio, Y.: Deep sparse rectifier neural networks. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 315–323 (2011)
16. Goodfellow, I., Bengio, Y. and Courville, A.: Deep Learning. MIT Press (2016)
17. Guss, W.H., Salakhutdinov, R.: On characterizing the capacity of neural networks using algebraic topology (2018). arXiv preprint arXiv:1802.04443
18. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
19. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
20. Kaul, P., Lall, B.: Riemannian curvature of deep neural networks. IEEE Trans. Neural Netw. Learn. Syst. (2019). https://doi.org/10.1109/TNNLS.2019.2919705
21. Kearns, M.J., Vazirani, U.V., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press (1994)
22. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
23. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
24. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
25. LeCun, Y., Cortes, C.: MNIST handwritten digit database (2010)
26. Lee, J.M.: Riemannian Manifolds: An Introduction to Curvature, vol. 176. Springer, New York (1997)
27. Lee, J.M.: Introduction to Smooth Manifolds, vol. 218. Springer, New York (2013)
28. Mallat, S.: Group invariant scattering. Commun. Pure Appl. Math. **65**(10), 1331–1398 (2012)
29. Mallat, S.: Understanding deep convolutional networks. Phil. Trans. R. Soc. A **374**(2065), 20150203 (2016)
30. Mathworks. im2col. https://in.mathworks.com/help/images/ref/im2col.html. Accessed 10 Feb 2019
31. Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., Bronstein, M.M.: Geometric deep learning on graphs and manifolds using mixture model CNNs. In: Proceedings of CVPR, vol. 1, p. 3 (2017)
32. Munkres, J.R.: Topology. Prentice Hall (2000)
33. Nakahara, M.: Geometry, Topology and Physics. CRC Press (2003)
34. Poole, B., Lahiri, S., Raghu, M., Sohl-Dickstein, J., Ganguli, S.: Exponential expressivity in deep neural networks through transient chaos. In: Advances in Neural Information Processing Systems, pp. 3360–3368 (2016)
35. Raghu, M., Poole, B., Kleinberg, J., Ganguli, S., Sohl-Dickstein, J.: Survey of expressivity in deep neural networks (2016). arXiv preprint arXiv:1611.08083
36. Ren, S., He, K., Girshick, R., Sun, J.: Faster r-cnn: towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)
37. Saxe, A.M., McClelland, J.L., Ganguli, S.:Exact solutions to the nonlinear dynamics of learning in deep linear neural networks (2013). arXiv preprint arXiv:1312.6120

38. Schutz, B.: A First Course in General Relativity. Cambridge University Press (2009)
39. Sermanet, P., LeCun, Y.: Traffic sign recognition with multi-scale convolutional networks. In: Neural Networks (IJCNN), pp. 2809–2813. IEEE (2011)
40. Shuman, D.I., Narang, S.K., Frossard, P., Ortega, A., Vandergheynst, P.: The emerging field of signal processing on graphs: extending high-dimensional data analysis to networks and other irregular domains. IEEE Signal Process. Mag. **30**(3), 83–98 (2013)
41. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
42. Topaz, C., Ziegelmeier, L., Halverson, T.: Topological data analysis of biological aggregation models. PloS one. **10**. https://doi.org/10.1371/journal.pone.0126383
43. Vapnik, V.N., Chervonenkis, A.Y.: On the uniform convergence of relative frequencies of events to their probabilities. In: Measures of Complexity, pp. 11–30. Springer, Cham (2015)
44. Wiatowski, T., Bölcskei, H.: A mathematical theory of deep convolutional neural networks for feature extraction (2015). arXiv preprint arXiv:1512.06293
45. Zomorodian, A., Carlsson, G.: Computing persistent homology. Discret. Comput. Geom. **33**(2), 249–274 (2005)

Scaling Analysis of Specialized Tensor Processing Architectures for Deep Learning Models

Yuri Gordienko, Yuriy Kochura, Vlad Taran, Nikita Gordienko, Alexandr Rokovyi, Oleg Alienin and Sergii Stirenko

Abstract Specialized tensor processing architectures (TPA) targeted for neural network processing has attracted a lot of attention in recent years. The computing complexity of the algorithmically different components of some deep neural networks (DNNs) was considered with regard to their further use on such TPAs. To demonstrate the crucial difference between TPU and GPU computing architectures, the real computing complexity of various algorithmically different DNNs was estimated by the proposed scaling analysis of time and speedup dependencies of training and inference times as functions of batch and image sizes. The main accent was made on the widely used and algorithmically different DNNs like VGG16, ResNet50, and CapsNet on the cloud-based implementation of TPA (actually Google Cloud TPUv2). The results of performance study were demonstrated by the proposed scaling method for estimation of efficient usage of these DNNs on this infrastructure. The most important and intriguing results are the scale invariant behaviors of time and speedup dependencies which allow us to use the scaling method to predict the

Y. Gordienko (✉) · Y. Kochura · V. Taran · N. Gordienko · A. Rokovyi · O. Alienin · S. Stirenko
National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, Ukraine
e-mail: yuri.gordienko@gmail.com

Y. Kochura
e-mail: iuriy.kochura@gmail.com

V. Taran
e-mail: vladtkv@gmail.com

N. Gordienko
e-mail: nik.gordiienko@gmail.com

A. Rokovyi
e-mail: alexandr.rokovoy@gmail.com

O. Alienin
e-mail: oleg.alenin@gmail.com

S. Stirenko
e-mail: sergii.stirenko@gmail.com

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_3

running training and inference times on the new specific TPAs without detailed information about their internals even. The scaling dependencies and scaling powers are different for algorithmically different DNNs (VGG16, ResNet50, CapsNet) and for architecturally different computing hardwares (GPU and TPU). These results give the precise estimation of the higher performance (throughput) of TPAs as Google TPUv2 in comparison to GPU for the large number of computations under conditions of low overhead calculations and high utilization of TPU units by means of the large image and batch sizes. In general, the usage of TPAs like Google TPUv2 is quantitatively proved to be promising tool for increasing performance of inference and training stages even, especially in the view of availability of the similar specific TPAs like TCU in Tesla V100 and Titan V provided by NVIDIA, and others.

Keywords Deep learning · Tensor processing architecture · Tensor processing unit · Tensor cores · Scaling · GPU · TPUv2 · Training time · Inference time · Speedup · MNIST · VGG16 · ResNet50 · CapsNet

1 Introduction

The current success of machine learning (ML), especially deep learning (DL) and deep neural networks (DNNs), is tightly coupled with the recent progress of high-performance computing (HPC) and specialized computing architectures. It allows using their intrinsic concurrency and specific abilities for better efficiency [1–4]. Along with a growth of the ML/DL problem complexity, size of datasets and DNNs, amount of features, classes and categories of processed objects increases. Therefore, the demands for the higher accuracy, quicker learning and inference is also tightly related to the demands for higher computing power, larger memory, wider network bandwidth, etc. For these purposes various computing infrastructures are used now: from desktop machine (with shared memory model) to parallel hardware architectures like high-performance computing cluster (with shared and distributed memory models) and up to cloud-based systems (distributed memory model), which are especially popular and widely used now.

The progress of these infrastructures follows the current trend of hardware acceleration for DL applications. During the last decade it was deeply connected with development of graphics processing units (GPU) as general purpose processors (GPGPU). However recently the wide variety of alternative old and new platforms appears: from the well-known digital signal processors (DSP) and field programmable gate-arrays (FPGA) [5, 6] to the quite new application-specific integrated circuit (ASIC) architectures [7] including neuromorphic hardware [8, 9], tensor-processing architectures (TPA) like TCs (Tensor Cores) by NVIDIA [10] and TPUs (tensor processing units) by Google [11] targeted for specialized tensor processing tasks that are heavily used in machine learning applications.

For efficient usage of the modern complex DNNs on these computing infrastructures various aspects of both of them should be taken into account. They are especially important for the new computing architectures on the ML/DL cloud-based systems (like Clarifai, Google Cloud Vision, Rekognition, Polly, and Lex Amazon, Microsoft Azure Cognitive Services, IBM Watson, etc.) that become extremely popular in scientific community and commercial applications. The main problem of cloud-based solutions in comparison to localized solutions is the proper estimation of the training and inference times that is essential for commercial applications.

In this work, we shortly discuss some specialized TPA, emphasize their cloud-based implementations and demonstrate the results of performance study for some popular DNNs on such infrastructure, and propose the scaling method for estimation of efficient usage of these DNNs on this infrastructure.

The main aim of this paper is to investigate scaling of training and inference performance for the available GPUs and TPUs with an increase of batch size and image size that allows making predictions of running times and estimate the hidden overheads in proprietary TPA solutions even.

The remainder of this chapter is organized as follows. In Sect. 2 we give the brief outline of the state of the art in tensor processing and TPAs used. Section 3 contains the description of the experimental part related with the selected hardware, dataset, models, and metrics used. Section 4 reports about the experimental results obtained, Sect. 5 is dedicated to discussion of these results, and Sect. 6 summarizes the lessons learned.

2 Background and Related Work

In contrast to the traditional HPC tasks the current ML/DL tasks are mostly resolved by means of the specific accelerated systems like GPU and FPGA despite their usage for HPC before. The essence of these specialized architectures is in their intrinsic abilities to use the high data parallelism efficiently. Therefore their usage becomes de facto standard in ML/DL applications nowadays. Despite the evident progress, even single-machine nodes accelerated by GPU cards cannot satisfy the quickly growing computing demands from ML/DL problems. As a response to these new requirements for higher computing power the following mainstream trends appear: multi-node architectures (in cluster and cloud systems), specialized TPAs, and their combinations [12].

The additional demand for the new specialized computing architectures is related with the usage of ML/DL application on various mobile devices from consumer types (like smartphones, tablets, wearable electronics, etc.) to industrial ones (like security systems, health monitoring, autonomous driving, etc.). This demand is tightly related with the strict requirements to the lower memory usage, the number of computing operations, and related energy waste [13–15]. To resolve these hardships specialized TPAs like TC [10] and TPU [11] appear and are widely used now in consumer and industrial implementations.

Recent trends in acceleration of DNNs for mobile devices with TPA can be classified into some categories: optimized implementation, quantization, and structured simplification that convert a DNN into a compact one [16, 17]. For example, structured simplification envelopes several approaches like tensor factorization [17], sparse connection [18], and channel pruning [19].

The extensive reviews on efficient processing of DNNs usually related with various general perspectives, models, optimization algorithms, and datasets. Others consider computation techniques for DNN components with regard to inherent parallelism of the targeted hardware using some techniques to reduce the overall memory used on the targeted hardware [19].

In this context, the strategic vision of parallelism applied for DNNs is of great importance for evaluation, implementation, and extension of algorithms and systems targeted at supporting distributed environments. Recently, some comparative measures on the approaches were discussed. Their concurrency and the average parallelism using the work-depth model were analyzed [12].

2.1 Tensor Cores

Tensor Cores (TCs) are hardware matrix math accelerators proposed in Volta architecture by NVIDIA. Tensor Cores provide a 4 × 4 × 4 matrix processing array which performs the operation $D = A * B + C$, where A, B, C and D are 4 × 4 matrices. TCs in combination with TensorRT library by NVIDIA allow us to perform several transforming and optimizing operations to DNN graph to improve performance. For example, they include elimination of layers with unused output to avoid unnecessary computation, fusion of some separate layers (convolution, bias, and activation) into a single layer, horizontal layer fusion by combining the functionally similar layers (with the same source tensor and the same operations with similar parameters), etc.

TC and TensorRT allow for using low-bit optimizations. For example, half-precision (also called FP16) arithmetic reduces memory footprint of DNNs and read-write overheads in comparison to FP32 or FP64 arithmetic. It enables deployment of larger DNNs and does it faster than FP32 or FP64 arithmetic. It is explained by the more efficient matrix operations on TCs, where the operation $D = A * B + C$ can be implemented by the inputs A and B which are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices. In addition, TensorRT automatically uses TCs for inference with FP16 arithmetic. As a result, TensorRT + TCs on Volta architecture V100 GPU card (with FP16 arithmetic) can significantly speed up inference time. For example, inference for ResNet-50 DNN model on GPU V100 + TensorRT + TCs (FP16) is ~4 × faster than inference with single precision (FP32) and ~8 × faster than inference with double precision (FP64) on the same card without TensorRT + TCs [20].

But it should be noted that these results were reported for the quite different batch sizes: 2 for V100 (FP32) and 16 for V100 + TensorRT + TCs (FP16), while our previous benchmark results demonstrated the huge influence of the batch size on the performance of some DNNs on GPU and TPU architectures [21].

Moreover, Turing architecture for TCs not only provide FP16/FP32 arithmetic like Volta architecture for Tensor Cores; they also add new INT8 and INT4 arithmetic. Usage of the lower bit representation on Tesla T4 GPU card with Turing architecture for TCs provides ~2.5 × faster inference in comparison to GPU V100 + TensorRT + TCs (FP16) for the same batch size 128 [22].

2.2 Tensor Processing Units

Recently Tensor Processing Units (TPUs) by Google attracted significant attention as another one TPA for increasing the efficiency and speed of DNNs. TPU contains 256 × 256 = total 65,536 arithmetic logic units (ALUs) that can process 65,536 multiply-and-add operations for 8-bit integers every cycle. As far as the TPU runs at 700 MHz, it can compute 65,536 × 7×108 = 46 × 1012 multiply-and-add operations or 92 × 1012 per second [11]. According to the announced benchmarks, TPU can process DNNs up to 30 × faster and can be up to 80 × more energetically efficient than CPUs or GPUs [11]. It is possible, because the TPU is specifically adapted to process DNNs with more instructions per cycle than CPU and GPU. Despite availability of some performance tests and extensive reviews of the available TPAs, like Google TPU versus NVIDIA GPU K80 [11] and Google Cloud TPUv2 versus GPU NVIDIA V100 [23], the systematic studies on their performance with regard to scaling (different dataset sizes, different batch sizes, etc.) are absent except for the several previous attempts [21].

Moreover TPU is the unique TPA because of its current availability as a cloud service resource only in the following configurations: Cloud TPU v2 (180 teraflops, 64 GB High Bandwidth Memory (HBM)), Cloud TPU v3 (420 teraflops, 128 GB HBM), Cloud TPU v2 Pod Alpha (11.5 petaflops, 4 TB HBM, 2-D toroidal mesh network) (https://cloud.google.com/tpu/) [11].

2.3 Other DNNs Accelerators

Despite the extremely high interest from various manufacturers to the new TPAs targeted for acceleration of DNNs, they are mostly proprietary solutions without details on their implementation, with scarce documentation and available application programming interfaces. Nevertheless, they become integral parts of computing devices as DNN accelerating co-processors, system on chips, or ASICs for specific applications (like computer vision or DSP) and the detailed reviews with benchmarks can be found elsewhere [24–28].

2.4 Parallel Algorithms and Tensor Processing Architectures

One of the ways to achieve the highest performance in GPU computing is to hide the long latency and other computational overheads by high data-level parallelism to achieve a high throughput, for example by the high batch size values [29, 30]. Usually, batch sampling is implemented by shuffling the whole dataset, and an entire pass over the dataset is called an epoch. As far as a full training procedure usually consists of tens to hundreds of such epochs, batch sampling with the largest possible batch can provide significant parallelization and throughput [31, 32].

Aiming on the highest accuracy the optimal batch size represent a tradeoff between minimal batch size 1 (one sample at each iteration) for traditional gradient descent, which is proven to converge, and the maximal batch (the whole dataset at each iteration), when convergence is not always proven to exist [33].

However, the optimal batch size is a complex optimization problem, as far as it is limited by requirements of accuracy and efficiency. But it is empirically known that batch size should not be too small, nor should it be too large to provide convergence and generalization [31, 34–36].

Most of the operations in learning can be modeled as operations on tensors (typically tensors as a parallel programming model). Such operations are highly data-parallel and only summations introduce dependencies. That is why quantitative analysis of TPAs under real working conditions can bring to the light the details of DNNs and TPAs themselves.

Following the commonly accepted paradigm a DNN can be represented by a directed acyclic graph (DAG) where the vertices are the computations and the edges are the data flows. The computational parallelism in such graph can be characterized by two main parameters: the volume of work W, which corresponds to the total number of vertices, and the depth D, which is the number of vertices on any longest path in the DAG. Usually these parameters can characterize the computational complexity on a parallel system and obtain some previous estimation for running time. For example, the running time on a single processor is ~*W*, on an infinite number of processes is ~*D*, and the average parallelism ~*W/D* [12].

In addition to the tests on GPU [37–42], recently the thorough performance analysis of the Google TPU was performed with some attempts to estimate influence of hyper-parameters on performance for TPU also [11, 43]. In addition, this work is aimed to give the answer to some questions, namely, when it could be more efficient to use GPU or TPU during training and inference phases for datasets of various sizes and batch sizes. In the next section the short description of the used datasets, network, equipment, and measurement methods is given.

This especially important in the view of the great interest to the influence of hyper-parameters of deep neural networks (DNN) on their training runtime and performance [37, 38], especially with regard to the batch size, learning rate, activation functions, etc. [39–42].

Nevertheless these benchmarks do not take into account the computing complexity of the models and do not estimate training and inference with regard to the different object sizes (for example, image sizes in classification problems), batch size, dataset size, etc. For example, for the very big datasets, data supply cannot be provided from in-memory only, and other ways should be used like data generators for reading data from hard disk or network including pipelines and other techniques. From the practical point of view these estimations are crucial for the actual training and inference times for various configurations of production DNN applications. From the fundamental point of view, as it will be shown below, the benchmarks on batch and object size influence can bring to the light some specific features of TPAs and DNNs used in them.

2.5 Parallel Algorithms and Computing Complexity in DNNs

Below the DNNs that are used for supervised learning will be considered, i.e. for optimizing a DNN on a set of labeled samples (train data) in such a way that for the given sample (test data) the DNN would return a label with some probability. It is assumed that both the train and test data are different parts of the same dataset. In this work, we consider one of the types of supervised learning problems, namely classification task, where the goal is to identify which class a sample most likely belongs to, for example the computer vision tasks. Among various DNNs we selected several well-known representatives of convolutional neural networks (CNNs) with slightly different structures. In CNNs the neurons are grouped to layers of several kinds that are described below. In computer vision images are used as input and represented as a 4-dimensional tensor $N \times C{\times}H \times W$, where N is the batch size (the number of images in the batch), H—is the height of the image (image size), W—is the width of the image (image size), C is the number colors (color channels).

In the DNNs (actually CNNs) considered here, the number of characteristic features (channels), as well as the width and height of an image, differ from layer to layer due to application of various following operators (described below) [44, 45].

In a convolutional layer an 3D tensor-shape image x (i.e., a slice of the 4D batch tensor which represents batch, i.e. the set of images) is convolved by the convolution operators (kernels) C_{out} of size $C_{\text{in}} \times K_h \times K_w$, where C_{in} is the input of the layer, C_{out} is the output of the layer, K_h—is the height and K_w is the width of of the convolution kernel. In terms of computing complexity the work (the number of mathematical operations) performed in this layer is equal to:

$$W_{\text{conv}} = \mathrm{O}\big(N C_{out} C_{in} H_i W_i K_x K_y\big) \tag{1}$$

where index i for H_i and W_i means the number of layer because the height and width of the image can be changed due to other operators like pooling.

The pooling operator reduces an input tensor by the width and height dimensions (and subsequently the number of trainable parameters of the model). The general aim of pooling is to reduce the size of the model while emphasizing important features, because subsequent pooling on all convolutional layers enables learning high-level features that correspond to larger regions in the original data. From algorithmic point of view this operator increase the local work at each convolution layer by W_{pool} operations, but decrease the global work by decreasing H_i and W_i at each convolution layer also, usually by exponential law. It performs some operation on contiguous sub-regions of the reduced dimensions, such as calculation of maximum or average value with the following computing complexity:

$$W_{\mathrm{pool}} = \mathrm{O}(NC_{in}H_iW_i) \tag{2}$$

The batch normalization operator makes normalization of images in the same batch with a zero mean and a variance of one that enables reducing the covariate shift and improving convergence with the following computing complexity:

$$W_{\mathrm{bn}} = \mathrm{O}(NC_{in}H_iW_i) \tag{3}$$

It should be noted that numerous additional operators, such as striding, padding, and dilating can be applied which can modify H_i and W_i and slightly influence the work, but these peculiarities are out of this consideration.

The fully connected layer is implemented as a matrix-matrix multiplication and addition with the following computing complexity:

$$W_{\mathrm{fcn}} = \mathrm{O}(C_{out} \cdot C_{in} \cdot N) \tag{4}$$

Usually DNN is represented as some function composition, where some of aforementioned operators applied to results of the previous operator (direct composition), or some operators might reuse the results of the previous operator with the output values of the more distant layer in multiple subsequent layers, forming shortcut connections like in residual networks like ResNet [46] or dense networks [47].

The whole number of computations in DNN includes computations from many layers with complex relationships and data workflows, especially taking into account the complex computations for the forward evaluation and backpropagation at different layer types. Sometimes the work performed in each layer can asymptotically dominate and allow for rough estimation of its computing costs. As far as DNN layers deal with 4-dimensional tensors and many operations localized inside them, they can be implemented for parallel execution at layer level. In this context, the runtime of a single DNN operator is hard to estimate even, despite some attempts to measure the performance of the highly-tuned matrix multiplication implementations [48]. In other works the runtime of some DNNs was estimated for batches of various sizes with ~5–20% error for GPUs [48] and CPUs [49, 50] in distributed environments even. Some performance models for DNNs were also proposed for operation counts with 10–30% error [51], and to estimate communication requirements for the convolution and pooling operators [52].

However, in general, the computing cost for the whole DNN is hard to estimate due to intrinsic complexity of most DNNs, especially taking into account different hardware architectures with the different available parallelism like TPAs. But from practical point of view it is especially important to use the intrinsic data parallelism in TPAs by increasing batch size and predicting the impact of batch size on the training and inference time. The fact is that most of the layer operators in DNNs are independent with respect to the number of samples (for example, images for CNNs) in the batch and direct parallelization way is to partition the work of the batch samples among multiple computational resources (for example, GPU cores and devices, or TPU cores, chips, pods, devices, and hosts).

Several attempts were made to check the reliability and feasibility of the general intuition that a bigger batch size will lead to better performance without losing considerable accuracy [31, 35, 36, 53, 54].

In this work the results on estimation of training and inference time for several DNNs are proposed on the basis of scaling approach that allow to use scaling method not only for runtime prediction for various batch and object size, but also for analysis of the hidden overheads for some new computing architectures, especially based on proprietary solutions with a limited notion about their internal organization on example of Google Cloud TPU.

3 Experimental and Computational Details

3.1 Datasets, Equipment, Metrics, and Models

Datasets: The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits (28 × 28 images) that become a standard benchmark for learning, classification and computer vision systems [55]. It was derived from a larger dataset known as the NIST Special Database 19 which contains digits, uppercase and lowercase handwritten letters. The subsets of these datasets were used with the maximally possible batch size (for the better runtime) starting from 8 images and up to 60,000 images.

Equipment: *GPU and TPU*. The GPU and TPU computing resources were used to investigate the influence of hardware-supported quantization on performance of the DNNs. NVIDIA Tesla K80 was used as GPU cards during these experiments as Google Collaborative cloud resources (https://colab.research.google.com). Google TPUv2 are arranged into 4-chip modules with a performance of 180 TFLOPS, and 64 of these modules are then assembled into 256 chip pods with 11.5 PFLOPS of overall performance. TPU 2.0 has an instruction set optimized for executing Tensorflow and capable of both training and running DNNs. A cloud TPUv2 version was used as a TPU-hardware during these experiments, where 8 TPU cores were available as Google Collaborative cloud resources also.

Deep Neural Networks: The following DNNs were used for this stage of research: VGG16 [56], ResNet50 [46], CapsNet (shown in Fig. 1) [57]. The idea behind them was to use the well-known DNNs, but use them for the standard MNIST dataset of moderate size and complexity to get results for reasonable period.

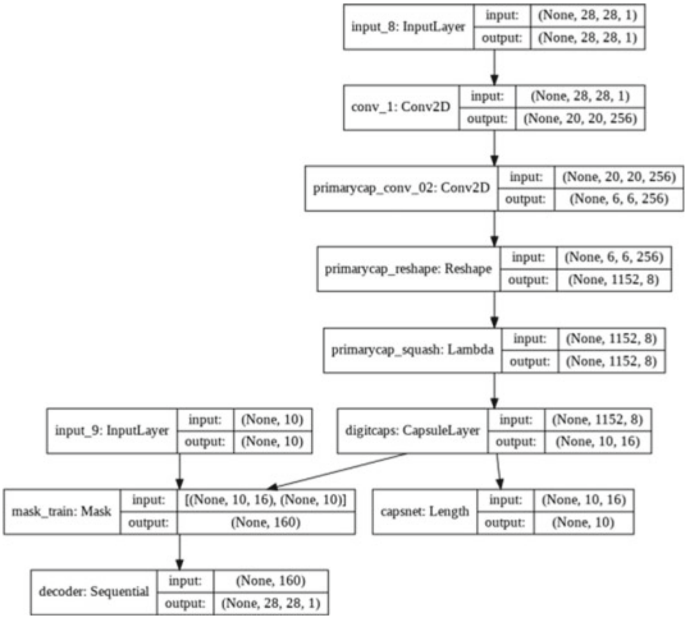


Fig. 1 The structure of the CapsNet deep neural network used in the work

VGG16 consists of 16 convolutional layers and has very uniform architecture with only 3 × 3 convolutions for all filters. It is widely described and used in the ML/DL community and currently is the most preferred choice for extracting features from images. The weight configuration of the VGG16 is publicly available and has been used in many other applications and challenges as a baseline feature extractor [56].

ResNet50 have the bigger depth and address the depth issue by training a slightly different inter-layer interaction. Instead of composing layers as in VGG16, it uses residuals implemented as “shortcut” identity connections to the network. The system then trains layers with respect to their residuals instead of their original values, and this solves the inherent degradation in accuracy as networks become deeper. With ResNet50 it became possible to train deeper networks with depths up to 152 layers and with further increase of the quality.

Recently the capsule network was proposed that is a nested set of neural layers that is different from the usual CNN. In the regular CNN more layers added in a stack, but in CapsNet more layers are added inside a single layer (or nested). The neurons inside a capsule encapsulate the above properties of one entity inside an image. Moreover, a capsule outputs a vector to represent the existence of the entity and the vector orientation represents the properties of the entity. The vector is sent to all possible parents in the neural network, and for each possible parent a capsule can find a prediction vector. Prediction vector is calculated based on the multiplication of it’s own weight and a weight matrix. Whichever parent has the largest scalar prediction vector product, increases the capsule bond. Rest of the parents decrease their bond. This routing by agreement method is superior to the current mechanism like max-pooling, because max pooling routes based on the strongest feature detected in the lower layer.

Metrics. Accuracy and loss values are calculated for training, validation, and inference phases, then receiver operating characteristic (ROC) curves are constructed and the area under curve (AUC) is calculated per class as their micro and macro averages. To emphasize the contribution of initialization phase for GPU and TPU, the following runtimes (both for GPU and TPU) per image were calculated for each run:

- time with overheads = the wall time of the *1*st iteration/number of images;
- time with overheads = the wall time of the 2nd iteration/number of images;
- time without overheads = the wall time of the 2nd iteration/number of images.

The speedup values were calculated as GPU runtimes divided by TPU runtimes.

During all trials the following actions were performed. Accuracy and loss values were calculated for training, validation, and inference phases (Fig. 2), then receiver operating characteristic (ROC) curves were constructed and the area under curve (AUC) were calculated per class as their micro and macro averages (Fig. 3). Below some examples of the training and validation histories are shown for GPU K-80 for MNIST dataset (Fig. 2) and the similar plots were obtained for TPUv2 MNIST (they are principally the same ones and not shown here because of the absence of difference).

The ROC-curves and AUC-values (Fig. 3) demonstrate the excellent prediction accuracy for 10 epochs even, which were used for comparison with the similar experiments on TPUv2.

Then training and testing (inference) times were calculated for different batch and image sizes and plotted as it is shown, for example, in Fig. 4 for GPU K80 and Fig. 5 for TPUv2. These results demonstrate the principally different scaling behavior of the training and testing (inference) times for these two different architectures.

The drastic difference of the 1st and 2nd iterations in comparison to the 3rd iteration means the availability of the starting data preparation and model compilation procedures (starting overheads stated as “overheads” below). They take place during

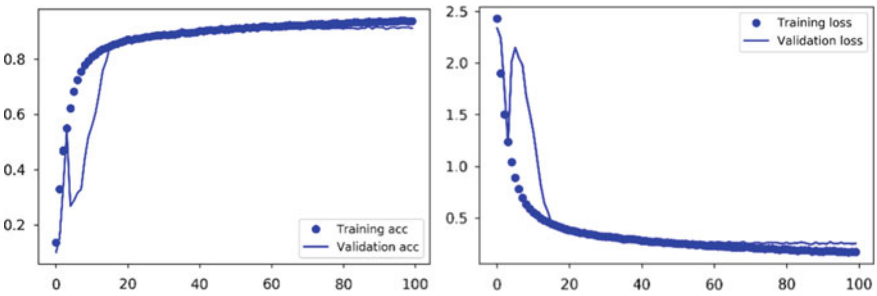


Fig. 2 Accuracy (left) and loss (right) during training and validation on GPU K80 for VGG16 (for the whole training part of MNIST dataset 60,000 images)

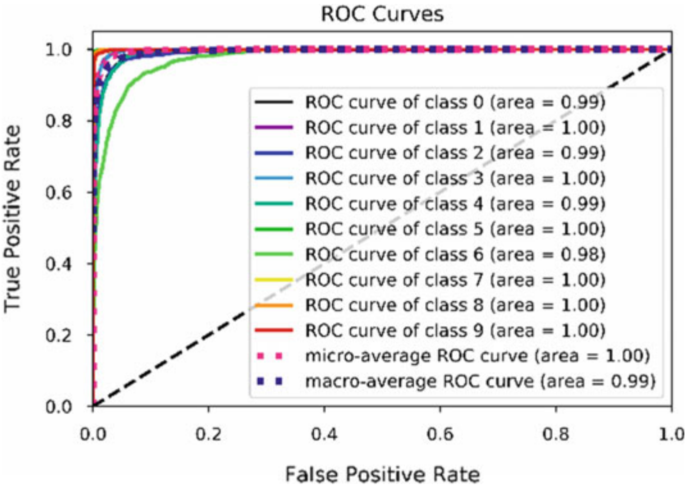


Fig. 3 ROC-curves and AUC-values for 10 classes on Google Cloud TPUv2 for ResNet50 (for the testing part of MNIST dataset—10000 images)

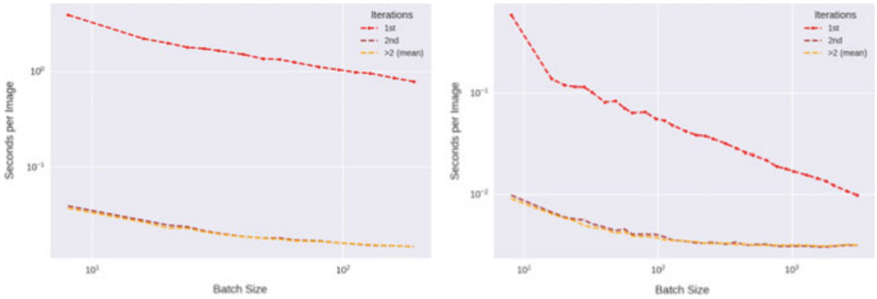


Fig. 4 Training (left) and testing (inference) (right) times versus batch size on GPU K80 (for the whole training part of MNIST dataset 60000 images)

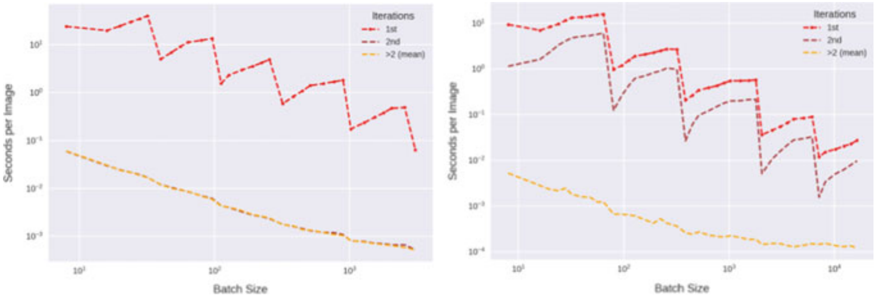


Fig. 5 Training (left) and testing (inference) (right) times versus batch size on Google TPUv2 (for the whole training part of MNIST dataset 60000 images)

the 1st iteration on GPU hardware, but during the 1st and 2nd iterations on TPU hardware and these overheads are much longer for TPU hardware. That is why the next scaling analysis was applied to these "overheads" and following iterations separately.

Then speedup values were also calculated and plotted as functions of the batch and image size.

The next section *Results* will focus on the training and inference times and speedups as functions of the batch and image size, because accuracy/loss histories, ROC-curves and AUC-values for all used DNNs on GPU-k80 and TPUv2 were the same in the limits of errors.

3.2 Computing Complexity of DNNs

The DNNs used in this work demonstrate the different increase of the computing complexity with increase of the input data size (side length of the input image). The effect is explained by the different algorithms implemented in them. The number of trainable parameters (w) and the correspondent theoretical numbers of floating point operations (FLOPs) (N) can be calculated manually or by means of the profiling functions provided in Tensorflow framework [58].

For example, the number of trainable parameters grows by different laws for VGG16: $w \sim s^{1/2}$ for a model with pooling layers and $w \sim s^2$ for a model without pooling layers (Fig. 6a). The correspondent theoretical number of floating point operations follow the same dependence: $N \sim s^{1/2}$ for VGG16 with pooling layers and $N \sim s^2$ for VGG16 without pooling layers (Fig. 6b).

As to ResNet50 the number of parameters was not changed in the used implementation and the correspondent number of floating point operations was constant (Fig. 7a). In contrast in CapsNet the number of trainable parameters grows as $w \sim s^2$ with the similar growth of the correspondent number of floating point operations (that

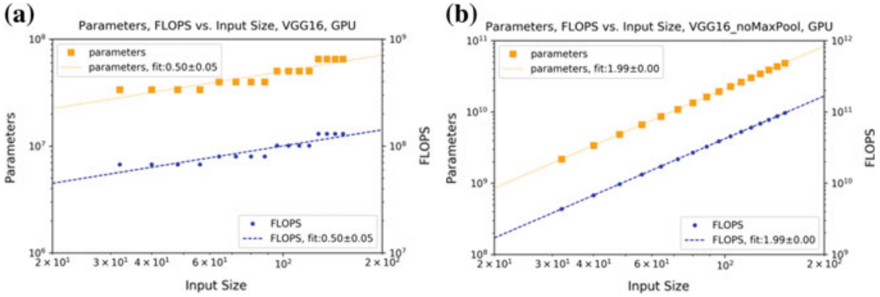


Fig. 6 The dependence of the number of parameters and floating point operations in VGG16 deep neural network as a function of the square image size (length)

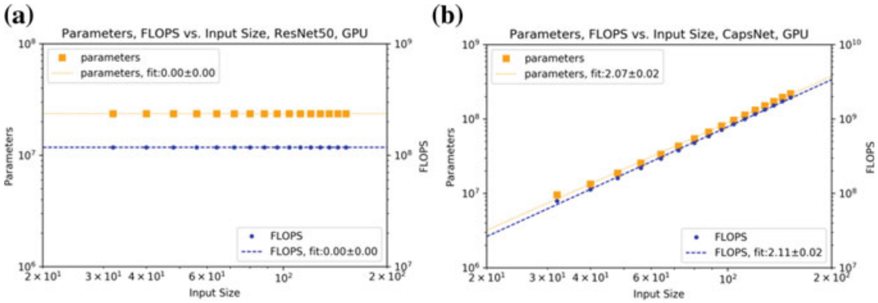


Fig. 7 The dependence of the number of parameters and floating point operations in VGG16 deep neural network as a function of the square image size ($H = W$)

is similar to VGG16 model without pooling layers when the most portion of calculations is performed in convolutional layers) (Fig. 7b).

As far as the real training and testing includes numerous supporting operations (like data processing, formatting, and so on) the related calculation overheads appear. These overheads can crucially change estimations of the wall times based on the numbers of floating point operations, because sometimes they are related with the organization of data manipulation, which can be different in various hardware like GPU and TPU. Moreover, they can be obscured from users, especially in cloud implementations (like in TPUv2 provided in Google Cloud).

But a comparison of the theoretical predictions with the real wall times as functions of input data size and batch size could provide real estimations and give some insights as to the principles of work of some hardware. That is why the further research was related to running numerous training and testing trials on MNIST dataset for already mentioned DNNs like VGG16, ResNet50, and CapsNet on GPU and TPU architectures. The main aim was to made comparative analysis of GPU and TPU on the basis of the time dependence and speedup versus different input data sizes and batch sizes. For this purpose the scaling analysis was used that is described in the next section.

3.3 Scaling Analysis

To demonstrate the main difference between these computing architectures (TPU and GPU) the real computing complexity for various algorithmically different DNNs was estimated by the scaling analysis. The scaling technique is widely used in various fields of science [59, 60], including finance [61], computer science and networks [62], biology [63], physics [64], materials science [65], geology [66], aggregation processes [67, 68], etc.

The idea behind the scaling technique is the assumption that the scale invariant behavior of some functions characterizes the process with regard to the change of

Table 1 The powers in scaling laws for the numbers of trainable parameters (α) and the theoretical numbers of floating point operations (β) for various DNN models

Model	α (parameters)	β (FLOPs)
VGG16	0.50 ± 0.05	0.50 ± 0.05
VGG16 (no MaxPool)	1.99 ± 0.01	1.99 ± 0.01
ResNet50	0.00 ± 0.00	0.00 ± 0.00
CapsNet	2.07 ± 0.02	2.11 ± 0.02

the other characteristics. It is assumed the proper re-normalization (scaling) of the functions allow us to find the general similarity for the functions characterizing the process under different values of scaling parameter.

Assuming that we have some function $f(x,\ldots)$ and change of its argument x by the scale factor k changes it by k^α times means

$$f(kx,\ldots) = k^\alpha f(x,\ldots) \tag{5}$$

that is typical for homogeneous functions. By such changes of several parameters for different systems (for example, different unknown hardware architectures) one can find the arguments for which this function can be scaled with insights as to the reasons for this scaling.

Returning to the previous section the numbers of trainable parameters for all networks demonstrate the similar following scaling laws:

$$w = f(s,\ldots) = s^\alpha f_{sc} \tag{6}$$

and

$$N = g(s,\ldots) = s^\beta g_{sc}, \tag{7}$$

where α and β are given in Table 1, f_{sc} and g_{sc} are scaled versions of f and g functions.

4 Results

4.1 Vgg16

To characterize the running time, the following notation will be used (for VGG16 here and all other DNNs below):

$$t_{regime} = f_{regime}(s, b, it), \tag{8}$$

where

t_{regime}—the wall running time in one of two regimes: training ($regime = train$) and testing or inference ($regime = inf$) regime;

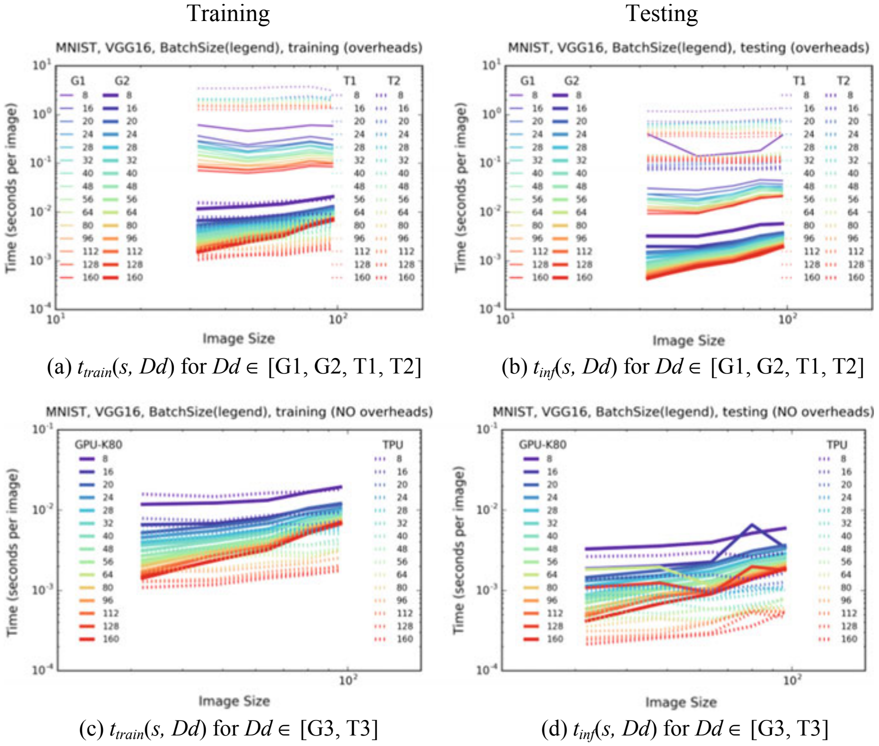
Dd—the running trials, where *D* is one of two devices: GPU K-80 ($D = \mathrm{G}$) and TPUv2 ($D = \mathrm{T}$), and *d* is the number of running iteration: 1 for 1st, 2 for 2nd, 3 for $d > 2$;

s—the image size (the side length of square images $H = W$ from MNIST dataset scaled from 28 × 28 to 96 × 96 pixels);

b—the mini-batch (batch) size from 8 to maximum possible number of images in a batch.

For example, $t_{\mathrm{inf}} = f_{\mathrm{inf}}(s, b, \mathrm{T1})$ means the inference time as a function image size *s* and batch size *b* for the 1st iteration on TPUv2 device.

The top images (Fig. 8a, b) represent the first and second iterations for GPU-K80 (G1 and G2) and TPU (T1 and T2) where significant overheads were observed, and the bottom images (Fig. 8c, d) represent the third iteration with much lower overheads.



(a) $t_{train}(s, Dd)$ for $Dd \in$ [G1, G2, T1, T2]

(b) $t_{inf}(s, Dd)$ for $Dd \in$ [G1, G2, T1, T2]

(c) $t_{train}(s, Dd)$ for $Dd \in$ [G3, T3]

(d) $t_{inf}(s, Dd)$ for $Dd \in$ [G3, T3]

Fig. 8 Time (per image) versus image size for training (left) and testing (inference) (right) regimes for VGG16. Each curve corresponds to the batch size and iteration denoted in the legend

The following steady and different behavior can be observed for both GPU and TPU architectures:

- TPU causes the much bigger overheads (Fig. 8a, b) for both training and testing (inference) regimes for the 1st (T1) and 2nd (T2) iterations in comparison to the correspondent 1st (G1) and 2nd (G2) iterations at GPU. This difference is especially high for the latency time at the 1st iteration.
- TPU demonstrates the much bigger overheads for the 1st iteration and 2nd iterations in comparison to the later iterations (Fig. 8c, d) (which have the same running times in the limits of the standard deviation), but GPU demonstrates the much bigger overheads for 1st iteration only.
- TPU causes the huge overheads for both training and testing (inference) regimes at the 1st (T1) and 2nd (T2) iterations in comparison to the 3rd (T3) iteration and in comparison to all iterations at GPU.
- For the later than 2nd iterations (Fig. 8c, d) TPU demonstrates the much lower running times in comparison to GPU.
- All time versus image size curves for each regime $t_{regime} = f_{regime}(s, b, it)$ are visually similar and that is why hypothesis that they are homogeneous functions is proposed and will be checked in the next section below.

Scaling for Time Dependencies

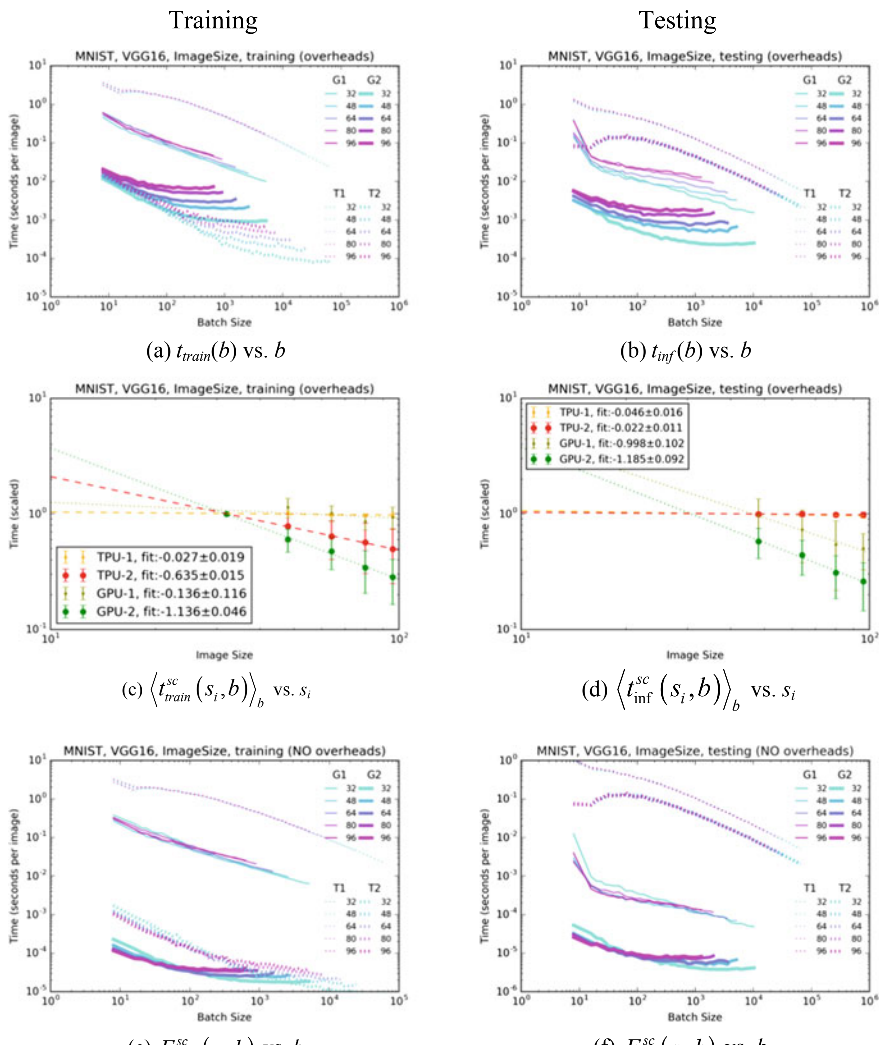
As far as the range of batch sizes was much bigger than the range of image sizes the scaling analysis was performed on the basis of the running time versus batch size curves. The running time versus batch size dependencies were plotted (Figs. 9a, b and 10a, b) with the very pronounced visual similarity of the curves for each regime.

To make the scaling analysis the following scaling procedure was applied:

$$\begin{aligned}
&t^{sc}_{regime}(s_i, b) = \frac{f_{regime}(s_i, b, it)}{f_{regime}\left(s_{\min}, b, it\right)} = \\
&= \frac{s_i^{\alpha} b^{\beta} f^{sc}_{regime}(o(s_i), o(b), it)}{s_{\min}^{\alpha} b^{\beta} f^{sc}_{regime}\left(o\left(s_{\min}\right), o(b), it\right)} = s_i^{\alpha} F^{sc}_{regime}(s_i, b), \\
&\text{where } F^{sc}_{regime}(s_i, b) = s_{\min}^{-\alpha} \frac{f^{sc}_{regime}(o(s_i), o(b), it)}{f^{sc}_{regime}\left(o\left(s_{\min}\right), o(b), it\right)}
\end{aligned} \tag{9}$$

Under assumption of the low correlation between s_i^{α} and $F^{sc}_{regime}(s_i, b)$, i.e. if the covariance between them $\mathrm{Cov}\left(s_i^{\alpha}, F^{sc}_{regime}(s_i, b)\right) = 0$, after averaging each curve $t^{sc}_{regime}(s_i, b)$ over b one can obtain:

$$\begin{aligned}
&\left\langle t^{sc}_{regime}(s_i, b)\right\rangle_b = \left\langle s_i^{\alpha} F^{sc}_{regime}(s_i, b)\right\rangle = \\
&= s_i^{\alpha}\left\langle F^{sc}_{regime}(s_i, b)\right\rangle_b + \mathrm{Cov}\left(s_i^{\alpha}, F^{sc}_{regime}(s_i, b)\right) \approx s_i^{\alpha} C_i, \\
&\text{where } C_i = \left\langle F^{sc}_{regime}(s_i, b)\right\rangle_b = const.
\end{aligned} \tag{10}$$



(a) $t_{train}(b)$ vs. b

(b) $t_{inf}(b)$ vs. b

(c) $\left\langle t^{sc}_{train}\left(s_i, b\right)\right\rangle_b$ vs. s_i

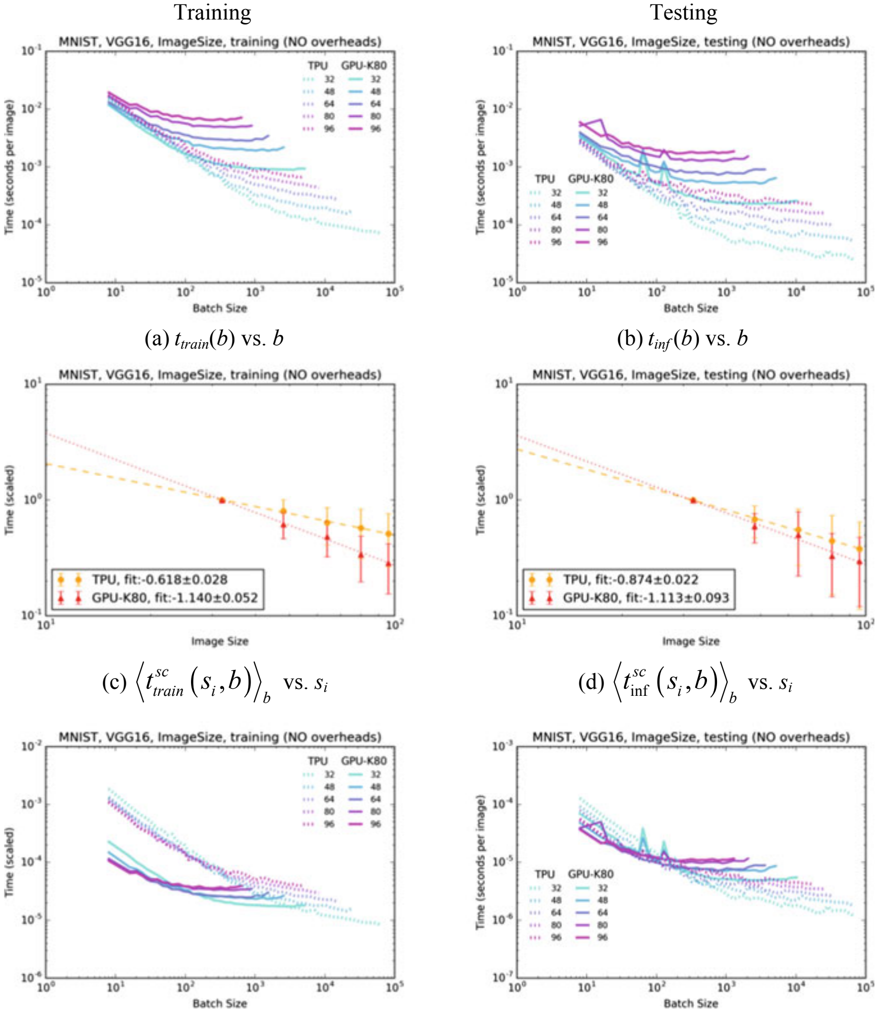
(d) $\left\langle t^{sc}_{\text{inf}}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(e) $F^{sc}_{train}\left(s_i, b\right)$ vs. b

(f) $F^{sc}_{\text{inf}}\left(s_i, b\right)$ vs. b

Fig. 9 Time per image for training (left) and testing (right) regimes for the 1st and 2nd iterations with big overheads, i.e. for $Dd \in$ [G1, G2, T1, T2] for VGG16

Then the power α can be determined after log-log plotting of the averaged values $\left\langle t^{sc}_{regime}(s_i, b)\right\rangle_b$ as a function of s_i for each curve from Figs. 9a, b and 10a, b and fitting the data by the direct line:



(a) $t_{train}(b)$ vs. b

(b) $t_{inf}(b)$ vs. b

(c) $\left\langle t^{sc}_{train}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle t^{sc}_{\inf}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(e) $F^{sc}_{train}\left(s_i, b\right)$ vs. b

(f) $F^{sc}_{\inf}\left(s_i, b\right)$ vs. b

Fig. 10 Time per image for training (left) and testing (right) regimes for 3rd iteration (without overheads) for VGG16

$$\alpha = \frac{\log\left(\left\langle t^{sc}_{regime}(s_i, b)\right\rangle_b\right) - \log(C_i)}{\log(s_i)} \tag{11}$$

The results of such fitting are shown for training regime on Figs. 9c and 10c, and for testing regime on Figs. 9d and 10d. If the aforementioned assumptions are true, then all points $\left\langle t^{sc}_{regime}(s_i, b)\right\rangle_b$ as a function of s_i in log-log plot should align along the direct line (Figs. 9c, d and 10c, d). Moreover, if the aforementioned assumptions are true, then all points for the functions $F^{sc}_{regime}(s_i, b)$ that are actually rescaled versions of $t^{sc}_{regime}(s_i, b)$ by division of s_i in the power α determined by fitting should collapse for each regime (Figs. 9e, f and 10e, f).

$$F^{sc}_{regime}(s_i, b) = \frac{t^{sc}_{regime}(s_i, b)}{s_i^{\alpha}} \tag{12}$$

Finally, both these conditions are true, namely, points are aligned along straight line (Figs. 9c, d and 10c, d) and the regime curves collapse (Figs. 9e, f and 10e, f), that confirm the previous assumptions and support the idea about scaling dependence:

$$t_{regime}(s) \sim s^{\alpha_r}$$

where α_r is the power characteristic for the specific regime.

Scaling for Speedup Dependences

The same scaling procedure can be applied to the speedup dependence versus image size and batch size for training and testing trials on TPU in comparison to the same trials on GPU:

$$\begin{aligned} S^{sc}_r(s_i, b) &= \frac{t_{Gd,r}(s_i, b)}{t_{Td,r}(s_i, b)} = \frac{s_i^{\alpha_{Gd,r}} b^{\beta_{Gd,r}} f^{sc}_{Gd,r}\big(o\big(s_{\min}\big), o(b), it\big)}{s_i^{\alpha_{Td,r}} b^{\beta_{Td,r}} f^{sc}_{Td,r}(o(s_i), o(b), it)} = \\ &= s_i^{\alpha_{Gd,r}-\alpha_{Td,r}} b^{\beta_{Gd,r}-\beta_{Td,r}} \frac{F^{sc}_{Gd,r}(s_i, b)}{F^{sc}_{Td,r}(s_i, b)} = s_i^{\alpha_{Sd,r}} b^{\beta_{Sd,r}} S^{sc}_{Sd,r}(s_i, b), \end{aligned} \tag{13}$$

where

$$\alpha_{Sd,r} = \alpha_{Gd,r} - \alpha_{Td,r} \tag{14}$$

$$\beta_{Sd,r} = \beta_{Gd,r} - \beta_{Td,r} \tag{15}$$

$$S^{sc}_{Sd,r}(s_i, b) = \frac{F^{sc}_{Gd,r}(s_i, b)}{F^{sc}_{Td,r}(s_i, b)}. \tag{16}$$

The values of the power $\alpha_{Dd,r}$ for VGG16 are summarized in Table 2.

Table 2 The image size powers $\alpha_{Dd,r}$ in scaling laws for the running time for VGG16

Time	$\alpha_{Td,r}$, TPU			$\alpha_{Gd,r}$, GPU		
	T1	T2	T3	G1	G2	G3
Testing	−0.05 ± 0.02	0.02 ± 0.01	−0.87 ± 0.02	−1.00 ± 0.10	**−1.19** ± 0.1	**−1.11** ± 0.09
Training	−0.03 ± 0.02	−0.64 ± 0.02	−0.62 ± 0.03	−0.14 ± 0.12	**−1.14** ± 0.05	**−1.14** ± 0.05

Table 3 The image size powers in scaling laws for the speedup: $\alpha^{f}_{Sd,r}$—fitted from the plots on speedup, and $\alpha^{pr}_{Sd,r}$—predicted from the scaling laws on running time for VGG16

	$\alpha^{pr}_{Sd,r}$=$\alpha_{Gd,r} - \alpha_{Td,r}$ predicted from time scaling			$\alpha^{f}_{Sd,r}$ fitted from speedup plots		
	S1	S2	S3	S1	S2	S3
Testing	−0.95 ± 0.10	−1.21 ± 0.09	−0.24 ± 0.09	−0.91 ± 0.11	−1.15 ± 0.08	−0.40 ± 0.07
Training	−0.11 ± 0.12	−0.50 ± 0.05	−0.52 ± 0.05	−0.29 ± 0.13	−0.70 ± 0.13	−0.78 ± 0.08

The values of the power $\alpha_{Sd,r}$ for VGG16 can be obtained by scaling analyses for time and speedup and compared in Table 3.

The batch size powers (β) in scaling laws for the speedup can be calculated directly from their plots (Figs. 11a and 12a, b), except for the testing regime with overheads (Fig. 12b) (Tables 4).

4.2 ResNet50

The top images (Fig. 13a, b) represent the first and second iterations for GPU-K80 (G1 and G2) and TPU (T1 and T2) where significant overheads were observed, and the bottom images (Fig. 13c, d) represent the third iteration with much lower overheads (Fig. 14).

The values of the power α for ResNet50 are summarized in Table 5.

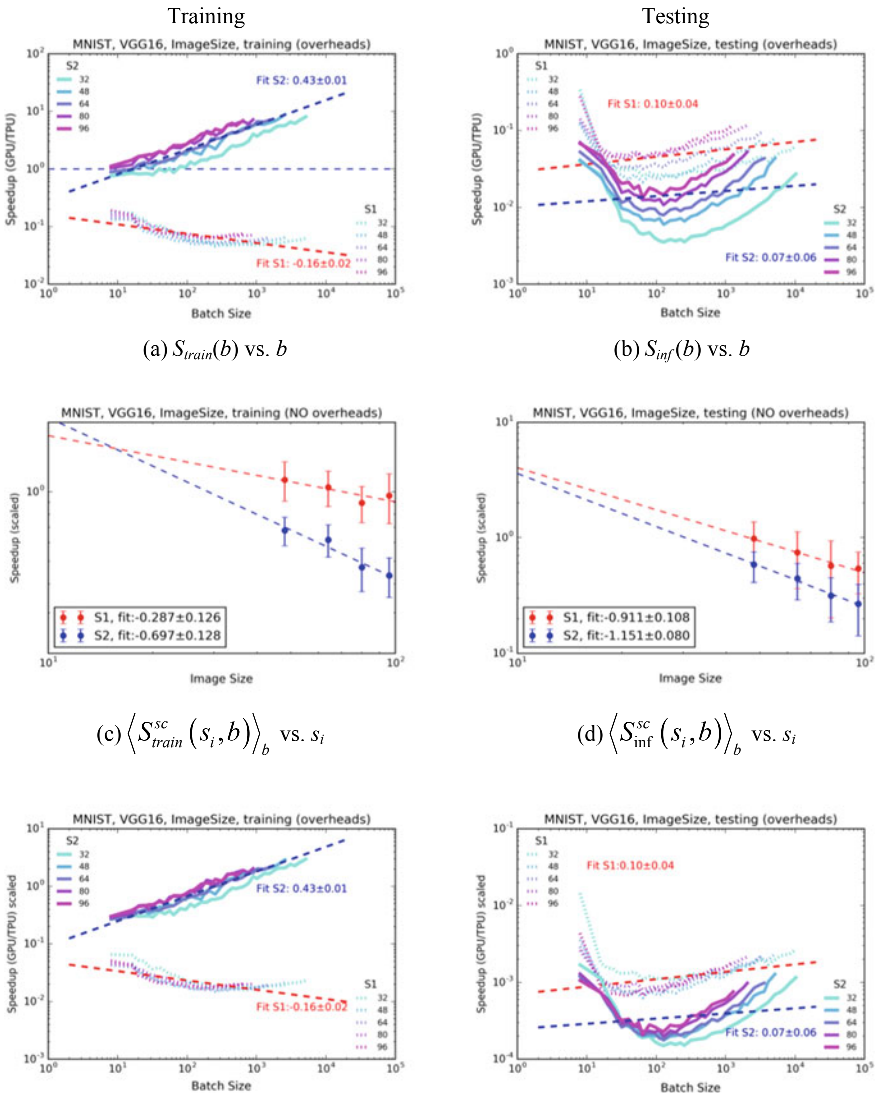
Just like in the case of VGG16 network, the batch size powers (β) in scaling laws for the speedup can be calculated directly from their plots (not shown here for brevity), except for the testing regime with overheads (Tables 6 and 7).

4.3 CapsNet

Again, the top images (Fig. 16a, b) represent the first and second iterations for GPU-K80 (G1 and G2) and TPU (T1 and T2) where significant overheads were observed, and the bottom images (Fig. 16c, d) represent the third iteration with much lower overheads (Fig. 17).

The values of the power α for CapsNet are summarized in Table 8.

Again, the batch size powers (β) in scaling laws for the speedup can be calculated directly from their plots (not shown here for brevity), except for the testing regime with overheads (Tables 9 and 10).



(a) $S_{train}(b)$ vs. b

(b) $S_{inf}(b)$ vs. b

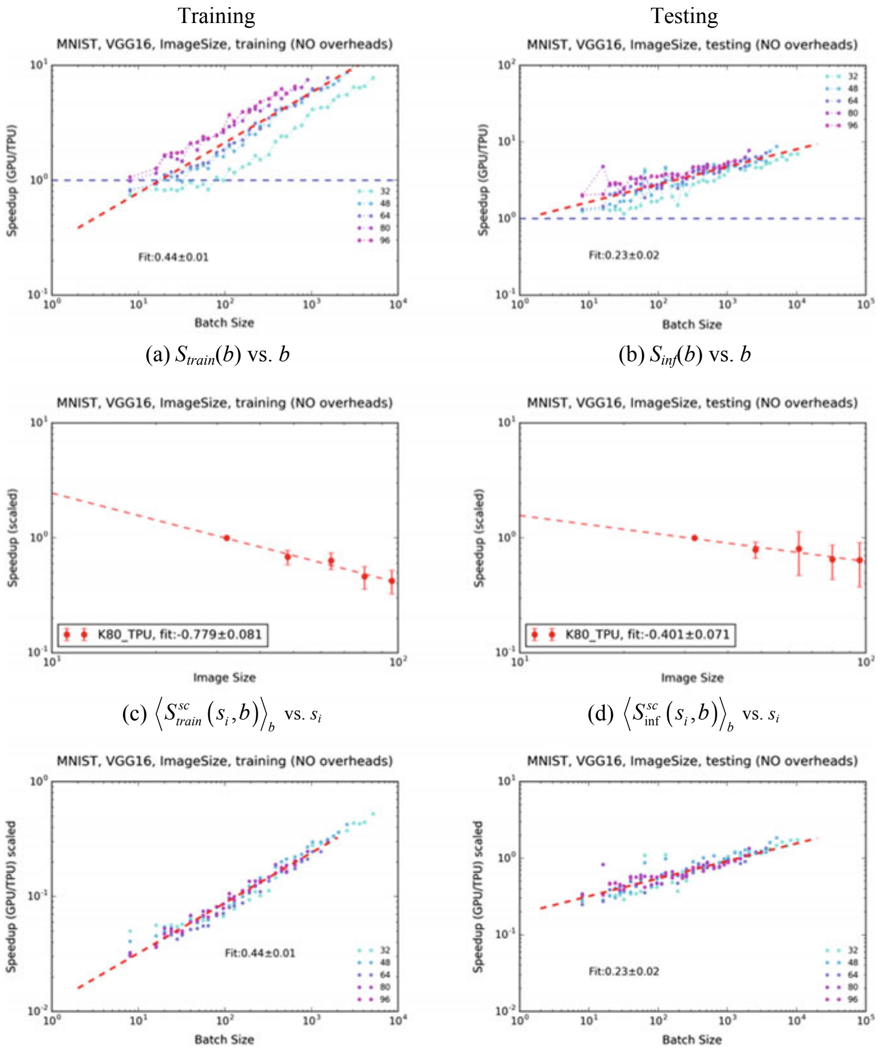
(c) $\left\langle S^{sc}_{train}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle S^{sc}_{\inf}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(e) $S^{sc}_{train}\left(s_i, b\right)$ vs. b

(f) $S^{sc}_{\inf}\left(s_i, b\right)$ vs. b

Fig. 11 Speedups for training (left) and testing (right) regimes for the 1st and 2nd iterations with big overheads, i.e. for $Dd \in$ [G1, G2, T1, T2] for VGG16



(a) $S_{train}(b)$ vs. b

(b) $S_{inf}(b)$ vs. b

(c) $\left\langle S^{sc}_{train}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle S^{sc}_{\inf}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(e) $S^{sc}_{train}\left(s_i,b\right)$ vs. b

(f) $S^{sc}_{\inf}\left(s_i,b\right)$ vs. b

Fig. 12 Speedups for training (left) and testing (right) regimes for 3rd iteration (without overheads) for VGG16

Table 4 The batch size powers (β) in scaling laws for the speedup for VGG16

Speedup	$\beta_{Sd,r}$, fitted from speedup plots		
	S1	S2	S3
Testing	0.10 ± 0.04	0.07 ± 0.06	0.23 ± 0.02
Training	0.16 ± 0.02	0.43 ± 0.01	0.44 ± 0.01

Table 5 The image size powers $\alpha_{Dd,r}$ in scaling laws for the running time for ResNet50

Time	$\alpha_{Td,r}$, TPU			$\alpha_{Gd,r}$, GPU		
	T1	T2	T2	G1	G2	G3
Testing	-0.33 ± 0.04	-0.36 ± 0.07	-0.59 ± 0.07	-0.47 ± 0.11	$-\mathbf{1.08} \pm 0.05$	$-\mathbf{1.19} \pm 0.05$
Training	-0.04 ± 0.30	-0.24 ± 0.01	-0.25 ± 0.02	0.03 ± 0.02	$-\mathbf{0.94} \pm 0.05$	$-\mathbf{0.97} \pm 0.04$

Table 6 The image size powers in scaling laws for the speedup: $\alpha^{f}_{Sd,r}$—fitted from the plots on speedup, and $\alpha^{pr}_{Sd,r}$—predicted from the scaling laws on running time for ResNet50

	$\alpha^{pr}_{Sd,r} = \alpha_{Gd,r} - \alpha_{Td,r}$ predicted from time scaling			$\alpha^{f}_{Sd,r}$ fitted from speedup plots		
	S1	S2	S3	S1	S2	S3
Testing	-0.14 ± 0.11	-0.72 ± 0.07	-0.6 ± 0.07	-0.38 ± 0.50	-1.31 ± 0.75	-0.76 ± 0.11
Training	0.07 ± 0.30	-0.7 ± 0.05	-0.72 ± 0.04	0.21 ± 0.08	-0.85 ± 0.04	-0.91 ± 0.04

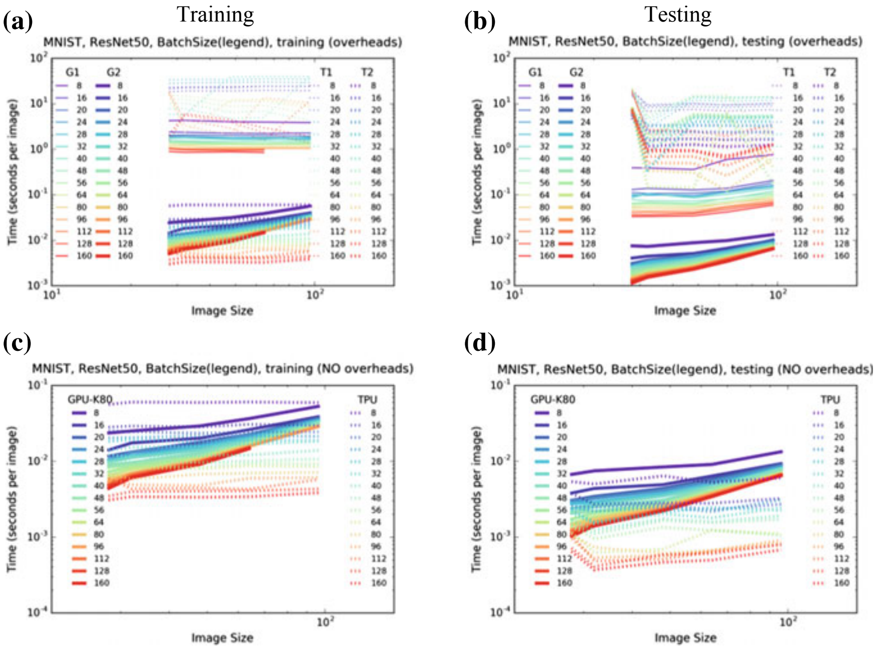
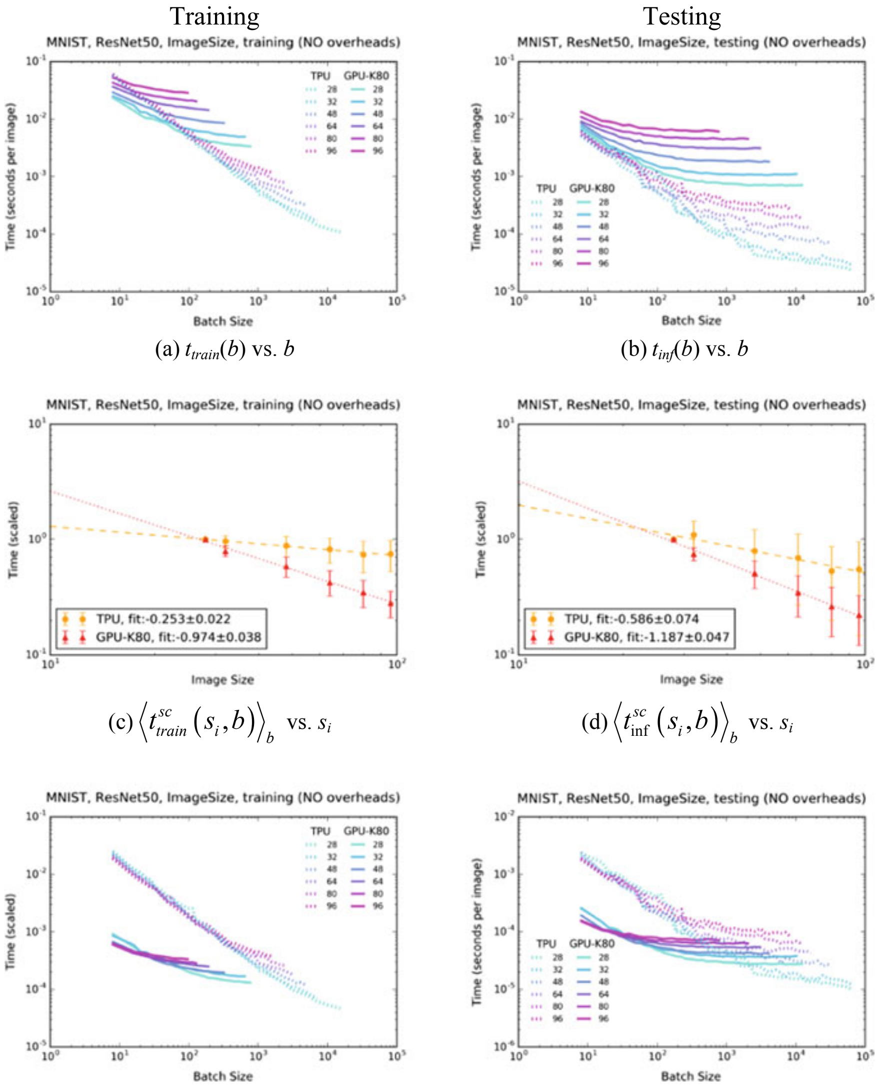


Fig. 13 Time (per image) versus image size for training (left) and testing (inference) (right) regimes for ResNet50. Each curve corresponds to the batch size and iteration denoted in the legend



(a) $t_{train}(b)$ vs. b

(b) $t_{inf}(b)$ vs. b

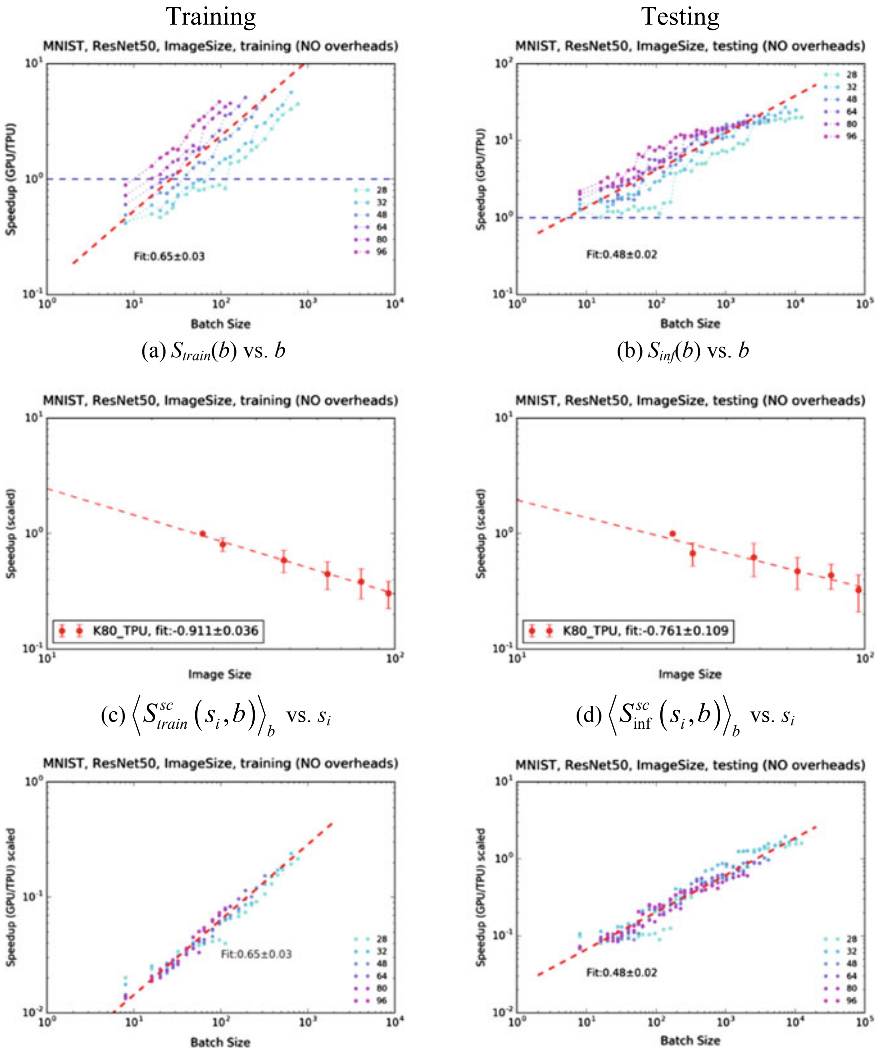
(c) $\left\langle t^{sc}_{train}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle t^{sc}_{\inf}\left(s_i, b\right)\right\rangle_b$ vs. s_i

(e) $F^{sc}_{train}\left(s_i, b\right)$ vs. b

(f) $F^{sc}_{\inf}\left(s_i, b\right)$ vs. b

Fig. 14 Time per image for training (left) and testing (right) regimes for 3rd iteration (without overheads) for ResNet50



(a) $S_{train}(b)$ vs. b

(b) $S_{inf}(b)$ vs. b

(c) $\left\langle S^{sc}_{train}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle S^{sc}_{\inf}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(e) $S^{sc}_{train}\left(s_i,b\right)$ vs. b

(f) $S^{sc}_{\inf}\left(s_i,b\right)$ vs. b

Fig. 15 Speedups for training (left) and testing (right) regimes for 3rd iteration (without overheads) for ResNet50

Table 7 The batch size powers (β) in scaling laws for the speedup for ResNet50

Speedup	$\beta_{Sd,r}$, fitted from speedup plots		
	S1	S2	S3
Testing	0.35 ± 0.09	0.71 ± 0.12	0.48 ± 0.02
Training	0.36 ± 0.18	0.63 ± 0.03	0.65 ± 0.03

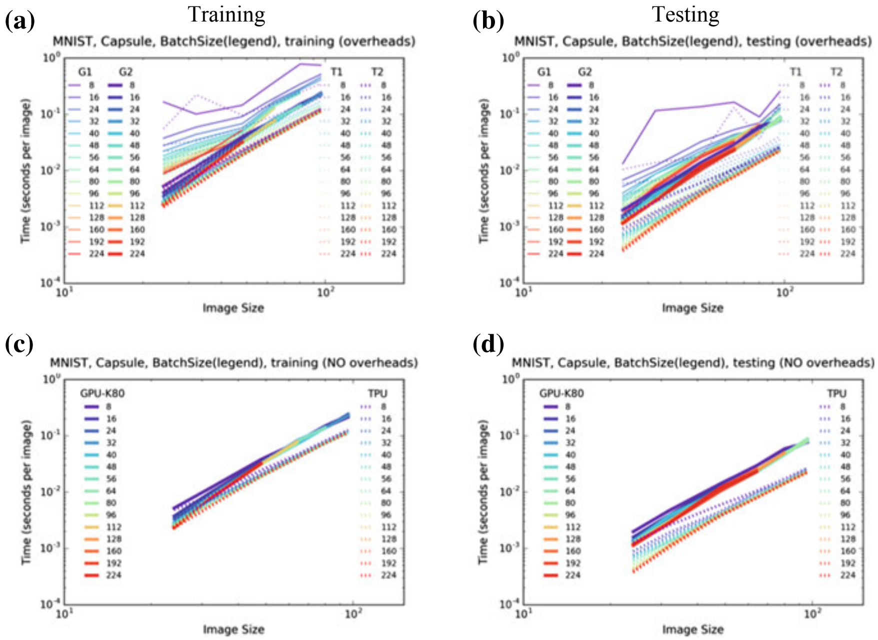


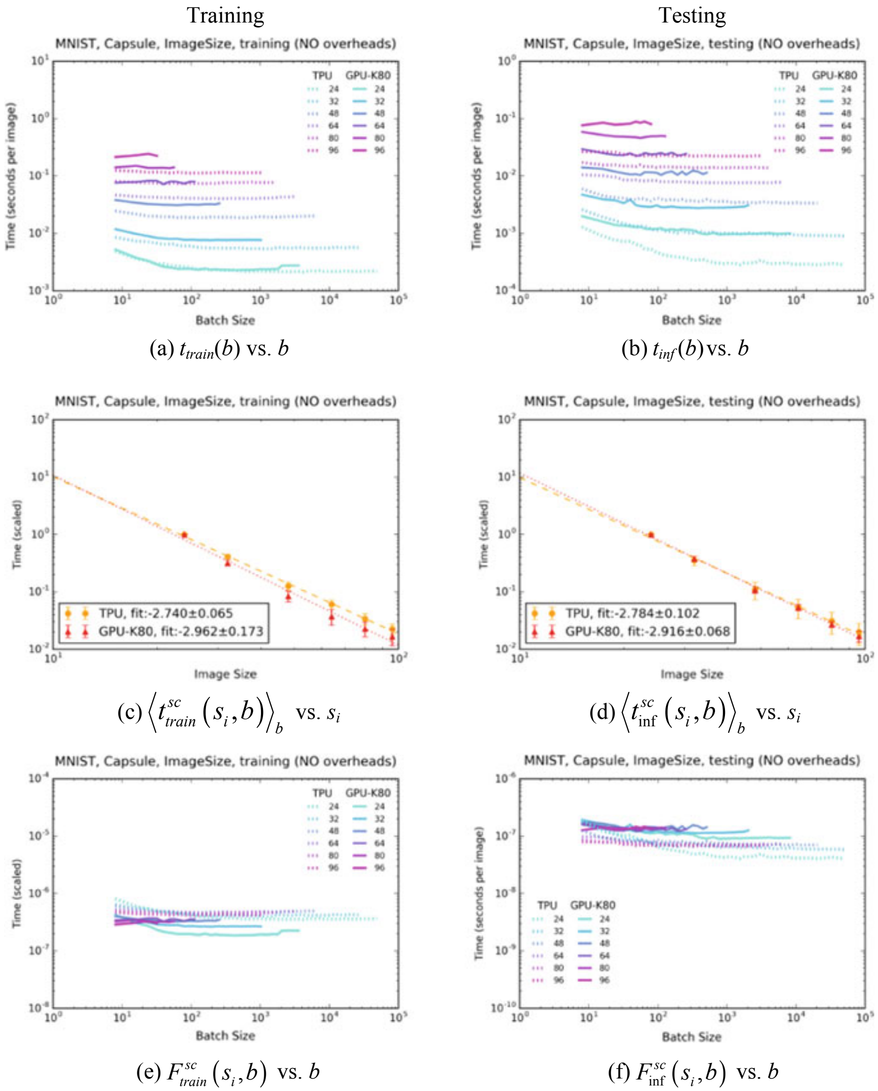
Fig. 16 Time (per image) versus image size for training (left) and testing (inference) (right) regimes for CapsNet. Each curve corresponds to the batch size and iteration denoted in the legend

5 Discussion

The significant speedup values for usage of TPU in comparison to GPU were obtained for quite algorithmically different models for the later iterations (>2nd) when the starting overheads do not have impact:

- VGG16—up to 10 × for training regime (Fig. 12a) and up to 10 × for testing regime (Fig. 12b),
- ResNet50—up to 6 × for training regime (Fig. 15a) and up to 30 × for testing regime (Fig. 15b),
- CapsNet—up to 2 × for training regime (Fig. 18a) and up to 4 × for testing regime (Fig. 18b).

These values were reached even for extremely low-scale usage of Google TPUv2 units (8 cores only) in comparison to the quite powerful GPU unit (NVIDIA Tesla K80). But the crucial difference between GPU and TPU architectures is the radically different values of latency time for specific data preparation and software compilation (much higher for TPU) before the 1st and 2nd iterations. This difference in the favor of GPU and that is why no speedup >1 was observed for all models at the 1st and 2nd iterations (Fig. 11).



(a) $t_{train}(b)$ vs. b

(b) $t_{inf}(b)$ vs. b

(c) $\left\langle t_{train}^{sc}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle t_{\mathrm{inf}}^{sc}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(e) $F_{train}^{sc}\left(s_i,b\right)$ vs. b

(f) $F_{\mathrm{inf}}^{sc}\left(s_i,b\right)$ vs. b

Fig. 17 Time per image for training (left) and testing (right) regimes for 3rd iteration (without overheads) for CapsNet

Table 8 The image size powers $\alpha_{Dd,r}$ in scaling laws for the running time for CapsNet

Time	$\alpha_{Td,r}$, TPU			$\alpha_{Gd,r}$, GPU		
	T1	T2	T3	G1	G2	G3
Testing	−2.21 ± 0.07	−2.78 ± 0.10	−2.78 ± 0.10	−2.30 ± 0.21	−**2.92** ± 0.06	−**2.92** ± 0.07
Training	−2.08 ± 0.03	−2.74 ± 0.06	−2.74 ± 0.07	−1.78 ± 0.14	−**2.95** ± 0.17	−**2.96** ± 0.17

Table 9 The image size powers in scaling laws for the speedup: $\alpha^{f}{}_{Sd,r}$—fitted from the plots on speedup, and $\alpha^{pr}{}_{Sd,r}$—predicted from the scaling laws on running time for CapsNet

	$\alpha^{pr}{}_{Sd,r} = \alpha_{Gd,r} - \alpha_{Td,r}$ predicted from time scaling			$\alpha^{f}{}_{Sd,r}$ fitted from speedup plots		
	S1	S2	S3	S1	S2	S3
Testing	−0.09 ± 0.21	−0.14 ± 0.10	−0.14 ± 0.10	−0.56 ± 0.13	−0.42 ± 0. 80	−0.42 ± 0.08
Training	0.3 ± 0.14	−0.21 ± 0.17	−0.22 ± 0.17	−0.36 ± 0.15	−0.42 ± 0.10	−0.44 ± 0.10

Table 10 The batch size powers (β) in scaling laws for the speedup for CapsNet

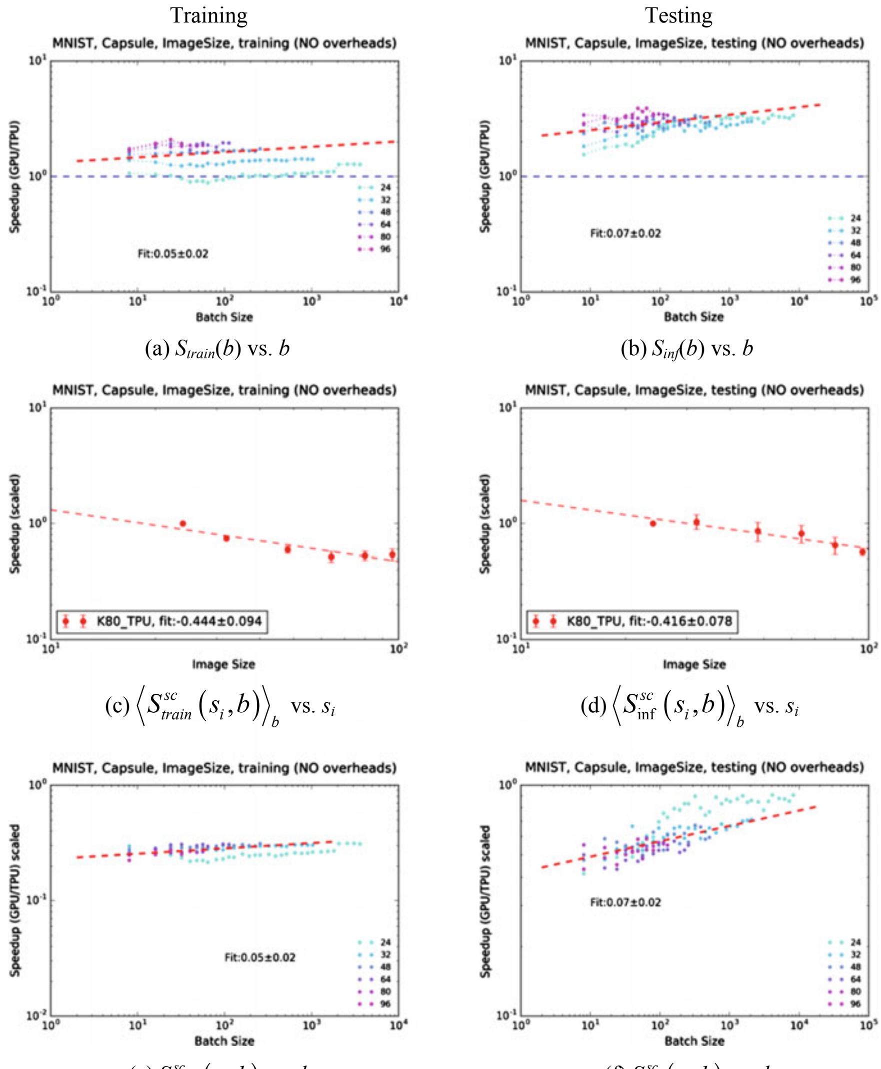
Speedup	$\beta_{Sd,r}$, fitted from speedup plots		
	S1	S2	S3
Testing	0.18 ± 0.05	0.07 ± 0.02	0.07 ± 0.02
Training	0.07 ± 0.03	0.03 ± 0.02	0.05 ± 0.02

The speedup values depend on the utilization level of TPUv2 units and increase start (i.e. speedup becomes >1) for different values of batch and image sizes for quite algorithmically different models for the later iterations (>2nd) when the starting overheads do not have impact:

- VGG16—for all batch and image sizes, except for the smallest image size and batch size <10 in training regime (Fig. 12a),
- ResNet50—for all batch and image sizes in testing regime, and for the various batch size in training regime, for example for $b > 10^2$ for the smallest image size (Fig. 15a),
- CapsNet—for all batch and image sizes, except for the smallest image size ($s = 24$) in training regime (Fig. 18a).

These results demonstrate that usage of TPAs as Google TPUv2 is more effective (faster) than GPU for the large number of computations under conditions of low calculation overheads and high utilization of TPU units. Moreover, these results were obtained for several algorithmically different DNNs without detriment to the accuracy and loss that were equal for both GPU and TPU runs up to the 3rd significant digit for MNIST dataset, and confirm the previous obtained similar results [21]. But it should be noted that these results were obtained without detriment to the accuracy and loss for the relatively simple MNIST dataset and low number of classes (=10). The current investigations of impact of batch, image, and network size even are under work now and their results will be published elsewhere [69].

The most important and intriguing results are the scale invariant behaviors of time and speedup dependencies which allow us to use this scaling method to predict the running times on the new specific architectures without detailed information about their internals even. The scaling dependencies and scaling powers are different for algorithmically different DNNs (VGG16, ResNet50, CapsNet) and for architecturally different computing hardware (GPU and TPU).



(a) $S_{train}(b)$ vs. b

(b) $S_{inf}(b)$ vs. b

(c) $\left\langle S^{sc}_{train}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(d) $\left\langle S^{sc}_{\inf}\left(s_i,b\right)\right\rangle_b$ vs. s_i

(e) $S^{sc}_{train}\left(s_i,b\right)$ vs. b

(f) $S^{sc}_{\inf}\left(s_i,b\right)$ vs. b

Fig. 18 Speedups for training (left) and testing (right) regimes for 3rd iteration (without overheads) for CapsNet

The crucial difference of the 1st and 2nd iterations in comparison to the 3rd iteration means the availability of the starting data preparation and model compilation procedures (starting overheads stated as “overheads” below). They take place during the 1st iteration on GPU hardware, but during the 1st and 2nd iterations on TPU hardware and these overheads are much longer for TPU hardware. The complexity of the scaled speedup dependencies for early iterations (1st and 2nd) and their relative simplicity for the late iterations (>2nd) reflect the sensitivity of DNNs to the initial stages of different computing hardware. This complexity is related with the complex and hidden (in proprietary TPUv2 hardware) details of data preparation and DNN compilation for TPA. The reasons of this complexity and sensitivity of DNNs to TPA should be the topic of future thorough investigations.

In addition to Google TPU architecture, the specific tensor processing hardware tools are available in the other modern GPU-cards like Tesla V100 and Titan V by NVIDIA based on the Volta microarchitecture with specialized Tensor Cores Units (640 TCU) and their influence on training and inference speedup are under investigation and will be reported elsewhere [69].

As far as the model size limits the available memory space for the maximum possible batch of images, other techniques could be useful for squeezing the model size, like quantization and pruning [70–72], and investigation of batch size increase on performance. These results can be used to optimize parameters of various ML/DL applications where a large batch of data should be processed, for example, in advanced driver assistance systems (ADAS), where specialized TPA-like architectures can be used [73].

6 Conclusions

In this work the short review is given for some currently available specialized tensor processing architectures (TPA) targeted on neural network processing. The computing complexity of the algorithmically different components of some deep neural networks (DNNs) was considered with regard to their further use on such TPAs.

To demonstrate the crucial difference between TPU and GPU computing architectures, the real computing complexity of various algorithmically different DNNs was estimated by the proposed scaling analysis of time and speedup dependencies of training and inference times as functions of batch and image sizes.

The main accent was made on the widely used and algorithmically different DNNs like VGG16, ResNet50, and CapsNet on the cloud-based implementation of TPA (actually Google Cloud TPUv2). The results of performance study were demonstrated by the proposed scaling method for estimation of efficient usage of these DNNs on this infrastructure.

The most important and intriguing results are the scale invariant behaviors of time and speedup dependencies which allow us to use the scaling method to predict the running training and inference times on the new specific TPAs without detailed information about their internals even. The scaling dependencies and scaling powers are different for algorithmically different DNNs (VGG16, ResNet50, CapsNet) and for architecturally different computing hardware (GPU and TPU).

These results give the precise estimation of the higher performance (throughput) of TPAs as Google TPUv2 in comparison to GPU for the large number of computations under conditions of low overhead calculations and high utilization of TPU units by means of the large image and batch sizes.

In general, the usage of TPAs like Google TPUv2 is quantitatively proved to be very promising tool for increasing performance of inference and training stages even, especially in the view of availability of the similar specific TPAs like TCU in Tesla V100 and Titan V provided by NVIDIA, and others.

References

1. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
2. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Netw. **61**, 85–117 (2015)
3. Wang, H., Raj, B.: On the origin of deep learning. arXiv preprint arXiv:1702.07800 (2017)
4. Bengio, Y.: Deep learning of representations: looking forward. In: International Conference on Statistical Language and Speech Processing, pp. 1–37. Springer, Berlin, Heidelberg (2013)
5. Lacey, G., Taylor, G.W., Areibi, S.: Deep Learning on FPGAs: Past, Present, and Future. arXiv preprint arXiv:1602.04283 (2016)
6. Nurvitadhi, E. et al.: Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In: Proceedings ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’17), pp. 5–14 (2017)
7. Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., Temam, O.: DianNao: a small-footprint high-throughput accelerator for ubiquitous machine learning. In: Proceedings 19th International Conference on ASPLOS, pp. 269–284 (2014)
8. Akopyan, F. et al.: TrueNorth: design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip. IEEE Trans. Comput. Aided Design Integr. Circuits Syst. **34**, 10 (2015), 1537–1557 (2015)
9. Ienne, P.: Architectures for Neuro-Computers: Review and Performance Evaluation. Technical Report. EPFL, Lausanne, Switzerland (1993)
10. In: NVIDIA Corporation. Programming Tensor Cores in CUDA 9, Accessed 2019. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9
11. Jouppi, N.P., et al.: In-datacenter performance analysis of a tensor processing unit. Int. Symp. Comput. Archit. **45**(2), 1–12 (2017)
12. Ben-Nun, T., Hoefler, T.: Demystifying parallel and distributed deep learning: an in-depth concurrency analysis. In: Computing Research Repository (CoRR) (2018)
13. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: MobileNets: efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861 (2017)
14. Real, E., Aggarwal, A., Huang, Y., Le, Q. V.: Regularized evolution for image classifier architecture search. arXiv:1802.01548 (2018)
15. Sze, V., Chen, Y.H., Yang, T.J., Emer, J.S.: Efficient processing of deep neural networks: a tutorial and survey. Proc. IEEE **105**(12), 2295–2329 (2017)
16. Han, S., Mao, H., and Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
17. Han, S., Pool, J., Tran, J., Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in Neural Information Processing Systems, pp. 1135–1143 (2015)
18. Mallya, A., Lazebnik, S. Packnet: Adding multiple tasks to a single network by iterative pruning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7765–7773 (2018)
19. Wang, E., et al.: Deep neural network approximation for custom hardware: Where We’ve Been, Where We’re Going. arXiv preprint arXiv:1901.06955 (2019)
20. Kama, S., Bernauer, J., Sharma, S.: TensorRT Integration Speeds Up TensorFlow Inference, Accessed 2019. https://devblogs.nvidia.com/tensorrt-integration-speeds-tensorflow-inference

21. Kochura, Y., Gordienko, Y., Taran, V., Gordienko, N., Rokovyi, A., Alienin, O., Stirenko, S.: Batch size influence on performance of graphic and tensor processing units during training and inference phases. In: Hu, Z. et al. (Eds.) Proceedings ICCSEEA 2019, AISC 938, pp. 1–11 (2019)
22. NVIDIA Corporation. NVIDIA AI inference platform, Accessed 2019. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/t4-inference-print-update-inference-tech-overview-final.pdf
23. Blog, R, Haußmann, E.: Comparing Google's TPUv2 against Nvidia's V100 on ResNet-50, Accessed 2019. https://www.hpcwire.com/2018/04/30/riseml-benchmarks-google-tpuv2-against-nvidia-v100-gpu
24. Qi, C.: Invited talk abstract: challenges and solutions for embedding vision AI. In: 2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2), p. 2. IEEE (2018)
25. Tsimpourlas, F., Papadopoulos, L., Bartsokas, A., Soudris, D.: A design space exploration framework for convolutional neural networks implemented on edge devices. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2212–2221 (2018)
26. Erofei, A.A., Druţa, C.F., Căleanu, C.D.: Embedded solutions for deep neural networks implementation. In: 2018 IEEE 12th International Symposium on Applied Computational Intelligence and Informatics (SACI) 000425-000430. IEEE (2018)
27. Seppälä, S.: Performance of neural network image classification on mobile CPU and GPU, Accessed 2019. https://aaltodoc.aalto.fi/bitstream/handle/123456789/31564/master_Seppälä_Sipi_2018.pdf
28. Ignatov, A., Timofte, R., Chou, W., Wang, K., Wu, M., Hartley, T., Van Gool, L.: Ai benchmark: Running deep neural networks on android smartphones. In: European Conference on Computer Vision, pp. 288–314. Springer, Cham (2018)
29. Zhu, H., Zheng, B., Schroeder, B., Pekhimenko, G., Phanishayee, A.: DNN-Train: Benchmarking and Analyzing DNN Training, Accessed 2019. http://www.cs.toronto.edu/ecosystem/papers/DNN-Train.pdf
30. Jäger, S., Zorn, H. P., Igel, S., Zirpins, C.: Parallelized training of Deep NN: comparison of current concepts and frameworks. In: Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning, pp. 15–20. ACM (2018)
31. Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., He, K.: Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677 (2017)
32. You, Y., Zhang, Z., Hsieh, C., Demmel, J.: 100-epoch ImageNet Training with AlexNet in 24 Minutes. arXiv:1709.05011 (2017)
33. Le, Q. V., Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Ng, A. Y.: On optimization methods for deep learning. In Proceedings 28th International Conference on Machine Learning, pp. 265–272 (2011)
34. Krizhevsky, A.: One weird trick for parallelizing convolutional neural networks. arXiv preprint arXiv:1404.5997 (2014)
35. Smith, S.L., Kindermans, P., Le, Q.V.: Don't decay the learning rate, increase the batch size. arXiv:1711.00489 (2017)
36. You, Y., Gitman, I., Ginsburg, B.: Large batch training of convolutional networks. arXiv:1708.03888 (2017)
37. Masters, D., Luschi, C.: Revisiting small batch training for deep neural networks. arXiv preprint arXiv:1804.07612 (2018)
38. Devarakonda, A., Naumov, M., Garland, M.: AdaBatch: adaptive batch sizes for training deep neural networks. arXiv preprint arXiv:1712.02029 (2017)
39. Smith, L. N.: A disciplined approach to neural network hyper-parameters: Part 1 - learning rate, batch size, momentum, and weight decay. arXiv preprint arXiv:1803.09820 (2018)
40. Kochura, Y., Stirenko, S., Alienin, O., Novotarskiy, M., Gordienko, Y., Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes. In: Advances in Intelligent Systems and Computing II. CSIT 2017. Advances in Intelligent Systems and Computing, 689, pp. 243–256. Springer, Cham (2017)

41. Kochura, Y., Stirenko, S., Gordienko, Y.: Comparative performance analysis of neural networks architectures on H2O platform for various activation functions. In: 2017 IEEE International Young Scientists Forum on Applied Physics and Engineering, pp. 70–73 (2017)
42. Kochura, Y., Stirenko, S., Alienin, O., Novotarskiy, M., Gordienko, Y.: Comparative analysis of open source frameworks for machine learning with use case in single-threaded and multi-threaded modes. In: 12th IEEE International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT), 1, pp. 373–376 (2017)
43. Jouppi, N., Young, C., Patil, N., Patterson, D.: Motivation for and evaluation of the first tensor processing unit. IEEE Micro **38**(3), 10–19 (2018)
44. Dumoulin, V., Visin, F.: A guide to convolution arithmetic for deep learning. arXiv:1603.07285 (2016)
45. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: Proceedings 32nd International Conference on Machine Learning, pp. 448–456 (2015)
46. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
47. Huang, G., Liu, Z., van der Maaten, L., Weinberger, K. Q.: Densely connected convolutional networks. In: Proceedings IEEE Conference on Computer Vision and Pattern Recognition (2017)
48. Oyama, Y., et al.: Predicting statistics of asynchronous SGD parameters for a large-scale distributed deep learning system on GPU supercomputers. In: IEEE International Conference on Big Data (Big Data), pp. 66–75 (2016)
49. Viebke, A., Memeti, S., Pllana, S., Abraham, A.: CHAOS: a parallelization scheme for training convolutional neural networks on Intel Xeon Phi. J. Supercomput. (2017)
50. Yan, F., Ruwase, O., He, Y., Chilimbi, T.: Performance modeling and scalability optimization of distributed deep learning systems. In: Proceedings 21st ACM International Conference on Knowledge Discovery and Data Mining, pp. 1355–1364 (2015)
51. Qi, H., Sparks, E.R., Talwalkar, A.: Paleo: a performance model for deep neural networks. In: Proceedings International Conference on Learning Representations (2017)
52. Demmel, J., Dinh, G.: Communication-optimal convolutional neural nets. arXiv:1802.06905 (2018)
53. Seide, F., Fu, H., Droppo, J., Li, G., Yu, D.: On parallelizability of stochastic gradient descent for speech DNNs. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 235–239 (2014)
54. Awan, A.A., Bedorf, J., Chu, C.H., Subramoni, H., Panda, D.K.: Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation. arXiv preprint arXiv:1810.11112 (2018)
55. LeCun, Y., Cortes, C., Burges, C.J.: MNIST handwritten digit database, Accessed 2019. http://yann.lecun.com/exdb/mnist
56. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: International Conference Learning Representations (2015)
57. Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: Advances in Neural Information Processing Systems, pp. 3856–3866 (2017)
58. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation, pp. 265–283 (2016)
59. Sornette, D.: Critical Phenomena in Natural Sciences: Chaos, Fractals, Self-organization and Disorder: Concepts and Tools. Springer Science & Business Media (2006)
60. Badii, R., Politi, A.: Complexity: Hierarchical Structures and Scaling in Physics, Vol. 6. Cambridge University Press (1999)
61. Mantegna, R.N., Stanley, H.E.: Econophysics: scaling and its breakdown in finance. J. Stat. Phys. **89**(1–2), 469–479 (1997)
62. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)

63. West, G.B., Brown, J.H., Enquist, B.J.: The origin of universal scaling laws in biology. Scaling Biology, 87–112 (2000)
64. Cardy, J.: Scaling and Renormalization in Statistical Physics, Vol. 5. Cambridge University Press (1996)
65. Gordienko, Y.G.: Molecular dynamics simulation of defect substructure evolution and mechanisms of plastic deformation in aluminium nanocrystals. Metallofiz. Noveishie Tekhnol. **33**(9), 1217–1247 (2011)
66. Torabi, A., Berg, S.S.: Scaling of fault attributes: a review. Mar. Pet. Geol. **28**(8), 1444–1460 (2011)
67. Gordienko, Y.G.: Change of scaling and appearance of scale-free size distribution in aggregation kinetics by additive rules. Physica A **412**, 1–18 (2014)
68. Gordienko, Y.G.: Generalized model of migration-driven aggregate growth—asymptotic distributions, power laws and apparent fractality. Int. J. Mod. Phys. B **26**(01), 1250010 (2012)
69. Yu, J., Tian, S.: A review of network compression based on deep network pruning. In: 3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019). Atlantis Press (2019)
70. Cheng, J., Wang, P.S., Li, G., Hu, Q.H., Lu, H.Q.: Recent advances in efficient computation of deep convolutional neural networks. Front. Inf. Technol. Electron. Eng. **19**(1), 64–77 (2018)
71. Zhu, M., Gupta, S.: To prune, or not to prune: exploring the efficacy of pruning for model compression. arXiv preprint arXiv:1710.01878 (2017)
72. Gordienko, Yu., Kochura, Yu., Taran, V., Gordienko, N., Bugaiov, A., Stirenko, S.: Adaptive iterative channel pruning for accelerating deep neural networks. XIth International Scientific and Practical Conference on Electronics and Information Technologies, Lviv, Ukraine, 16–18 September, 2019 (accepted)
73. Li, Y., Liu, Z., Xu, K., Yu, H., Ren, F.: A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks. ACM J. Emerg. Technol. Comput. Syst. **14**(2), 1–16 (2018)

Assessment of Autoencoder Architectures for Data Representation

Karishma Pawar and Vahida Z. Attar

Abstract Efficient representation learning of data distribution is *part and parcel* of successful execution of any machine learning based model. Autoencoders are good at learning the representation of data with lower dimensions. Traditionally, autoencoders have been widely used for data compression in order to represent the structural data. Data compression is one of the most important tasks in applications based on Computer Vision, Information Retrieval, Natural Language Processing, etc. The aim of data compression is to convert the input data into smaller representation retaining the quality of input data. Many lossy and lossless data compression techniques like Flate/deflate compression, Lempel–Ziv–Welch compression, Huffman compression, Run-length encoding compression, JPEG compression are available. Similarly, autoencoders are unsupervised neural networks used for representing the structural data by data compression. Due to wide availability of high-end processing chips and large datasets, deep learning has gained a lot attention from academia, industries and research centers to solve multitude of problems. Considering the state-of-the-art literature, autoencoders are widely used architectures in many deep learning applications for representation and manifold learning and serve as popular option for dimensionality reduction. Therefore, this chapter aims to shed light upon applicability of variants of autoencoders to multiple application domains. In this chapter, basic architecture and variants of autoencoder viz. Convolutional autoencoder, Variational autoencoder, Sparse autoencoder, stacked autoencoder, Deep autoencoder, to name a few, have been thoroughly studied. How the layer size and depth of deep autoencoder model affect the overall performance of the system has also been discussed. We also outlined the suitability of various autoencoder architectures to different application areas. This would help the research community to choose the suitable autoencoder architecture for the problem to be solved.

K. Pawar (✉) · V. Z. Attar (✉)
Department of Computer Engineering & IT, College of Engineering Pune (COEP), Pune, India
e-mail: kvppawar@gmail.com

V. Z. Attar
e-mail: vahida.comp@coep.ac.in

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_4

Keywords Autoencoders · Deep learning · Dimensionality reduction · Representation learning · Data representation

1 Introduction

Data representation plays a crucial role in designing a good model and eventual success of machine learning algorithms. In machine learning, representation learning allows a system to automatically discover the data representations required for feature detection from raw input data. Dimensionality of learnt representation of data is an important aspect since the performance of model tends to decline with increase in number of dimensions required for representing the distribution of data. Therefore, more emphasis has been given on the representation learning techniques by the researchers worldwide to perform feature learning and feature fusion resulting into compact and abstract representation of data [1].

A plethora of domain specific techniques have been evolved for learning the representation of data at compact and high level abstraction. The most conventional techniques such as Principal Component Analysis (PCA) and Latent Dirichlet Allocation (LDA) use linear transformation for data representation.

The autoencoders (AEs) are good at data denoising and dimensionality reduction. They work like dimensionality reduction technique such as PCA which project higher dimensional data to a lower dimensional space and preserve the salient features of data. However, PCA and autoencoders vary from each other in transformation they apply. PCA applies linear transformation whereas autoencoders apply non-linear transformations. Autoencoders are worse at compression than traditional methods like JPEG, MP3, and MPEG, etc. Since compression and decompression functions used in autoencoders are data-specific, autoencoders have problems generalizing to datasets other than what they trained on.

In the quest for Artificial Intelligence, deep learning has turned out to be the foremost solution applicable to solve complex problems in the domain of natural language processing [2], topic modeling [3, 4], object detection [5–7], video analytics [8, 9], image classification [10], prediction [11], etc. to mention but a few. Autoencoders have become the popular alternative for representation and manifold learning of data distribution in deep learning approaches. Due to this, many variants of autoencoder architectures have been put forth with specific trait applicable in unsupervised feature learning and deep learning.

This chapter gives detailed elaboration of what autoencoder is, taxonomy of autoencoders, domain-wise applications and factors regulating the working of autoencoders. The major contribution has been depicted in Fig. 1 as a central theme of this chapter and can be highlighted as follows.

- In-depth study of state-of-the-art autoencoder architectures and variants such as convolutional AE, regularized AE, variational AE, sparse AE, stacked AE, deep AE, generative AE, etc. has been performed in this chapter.

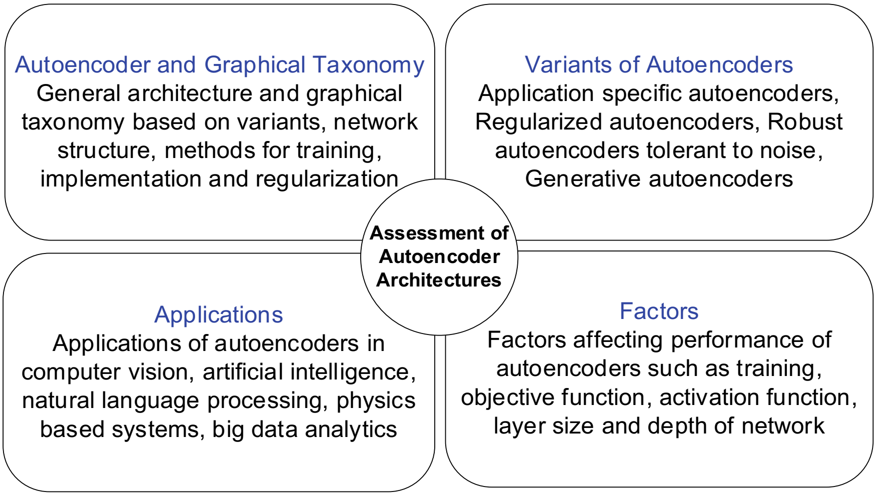

Autoencoder and Graphical Taxonomy
General architecture and graphical taxonomy based on variants, network structure, methods for training, implementation and regularization
Variants of Autoencoders
Application specific autoencoders, Regularized autoencoders, Robust autoencoders tolerant to noise, Generative autoencoders
Assessment of Autoencoder Architectures
Applications
Applications of autoencoders in computer vision, artificial intelligence, natural language processing, physics based systems, big data analytics
Factors
Factors affecting performance of autoencoders such as training, objective function, activation function, layer size and depth of network

Fig. 1 Central theme for assessment of autoencoder architectures

- The taxonomy of autoencoders corresponding to the factors required for designing them has also been put forth.
- The impact of layer size, depth on the performance of the model has been assessed.
- The applicability of variants of autoencoder architectures to different tasks has also been presented in the tabulated form.

The contents of this chapter are structured as follows. Section 2 gives overview of general architecture and proposed taxonomy of the autoencoders. The variants of autoencoders have been discussed in Sect. 3. Section 4 deals with factors such as training procedures, regularization strategies affecting the functionality of autoencoder based models. The characteristics of autoencoders along with their suitability for the application or task have been summarized in Sect. 5. Conclusion is mentioned in Sect. 6.

2 General Architecture and Taxonomy of Autoencoders

Autoencoders are self-supervised neural network architectures which are used to perform data compression in which compression and decompression functions are (i) lossy (ii) specific to data, and (iii) learnt from the data itself. For building an autoencoder, three things, namely, an encoding function, a decoding function, and a distance function are considered. Distance function is used for calculating the information loss between the compressed representation and the original input. The encoder and decoder are chosen to be parametric functions (expressed via neural networks), and to be differentiable by distance function. This enables the optimization

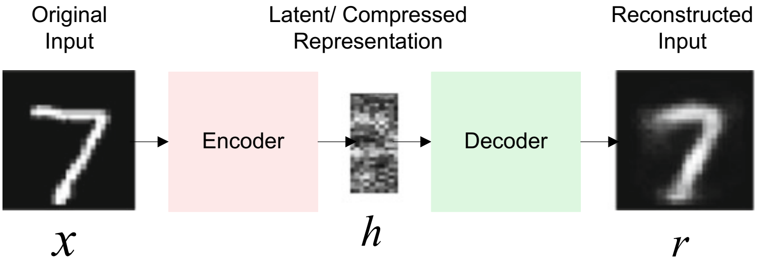


Fig. 2 General architecture of an autoencoder

of parameters in encoder and decoder functions. The reconstruction loss can be minimized using appropriate optimizers like Stochastic Gradient Descent (SGD).

The general idea of autoencoder is to pass an input data to an encoder to make compressed representation of the input. The encoding function can be represented as $h = f(x)$ where h is the latent representation. The middle layer, also known as "bottleneck layer," is the compressed representation of data from which original data can be reconstructed. The compressed representation h is passed to the decoder to get back the reconstructed data r. Decoding function is given by $r = g(h)$.

The encoder and decoder are both built using neural networks. The whole network is trained by minimizing the difference between input and output. There may be some loss of information due to fewer units. The whole autoencoder is represented mathematically by $g(f(x)) = r$. Figure 2 shows the general architecture of an autoencoder.

The loss function for autoencoder is given as

$$L = |x - g(f(x))| \tag{1}$$

where x is input such that $x \in \mathbb{R}^d$ and x is usually averaged over some input training set. The loss L penalizes the function $g(f(x))$ for being different from x. This loss L can be set as L^2 regularization of their difference. Keeping the size of latent representation small and choosing proper capacity of both encoding and decoding functions, any architecture of autoencoder can be trained well. Figure 3 depicts the taxonomy of autoencoders based on various factors to be considered for designing the autoencoders. The details of components depicted in Fig. 3 have been given in the sub-sequent sections of this chapter.

3 Variants of Autoencoders

Depending on the number of hidden layers present, autoencoders can be shallow or deep. Shallow AEs have input layer, single hidden layer and output layer. Deep AEs

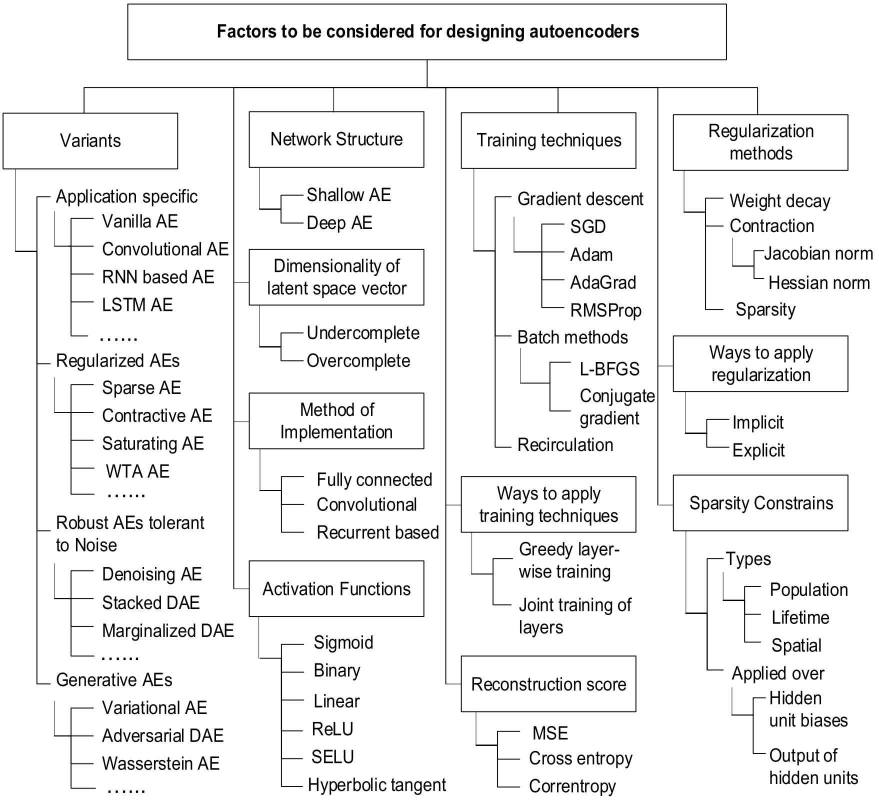


Fig. 3 Taxonomy of autoencoders

have multiple hidden layers. Based on dimensionality of latent space vector (representation) h, autoencoders can be classified as undercomplete and overcomplete autoencoders. It is more important and useful to train the autoencoder for performing the task of copying an input to output by constraining the latent representation. An autoencoder whose dimension of latent space vector h is less than dimension of input is called undercomplete. By undercomplete autoencoder, a model is forced to learn the most essential features of training data. If encoding and decoding functions are given too much capacity, then autoencoder will perform the task of copying the input to output without learning the essential features of data distribution. This issue arises when the dimensions of latent representation are equal to input dimensions, and in case of overcomplete autoencoders, dimensions of latent representation are greater than the input dimensions. The autoencoders can be implemented as fully connected, convolution based or recurrent based units. The variants of autoencoders have been discussed by following the categories as application specific AEs, regularized AEs, Robust AEs tolerant to noise and generative AEs. Many variants of autoencoders have been put forth till date under each of the mentioned categories. In this chapter, widely used autoencoders have been discussed.

3.1 Application Specific Autoencoders

Vanilla autoencoder

This is the simplest form of autoencoder having 3 layers of neural network viz. input layer for encoding, hidden layer representing the compressed representation and output layer for decoding. Figure 4 shows the architecture of vanilla autoencoder. Usually a single layer is not enough to learn the discriminative and representative features from input data. Therefore, researchers employ deep encoders (multilayer or deep) for better representation learning and dimensionality reduction. Hinton et al. [11] first proposed deep autoencoder for dimensionality reduction.

Deep autoencoder

Deep autoencoder [11] constitutes two symmetrical deep belief networks for encoding and decoding, each having 4–5 shallow layers. Restricted Boltzmann Machine (RBM) acts as the basic block of the deep belief network. Deep autoencoder works in 3 phases as pre-training, unrolling and fine-tuning.

Initially, a stack of RBMs having one layer for feature detection is used for pre-training in such a way that feature activations outputted by first RBM are used as input for the next RBM. These RBMs are unrolled after pre-training to get a deep autoencoder which can be fine-tuned using back-propagation. Figure 5 shows the design of deep autoencoder which is obtained by unrolling the stack of RBMs.

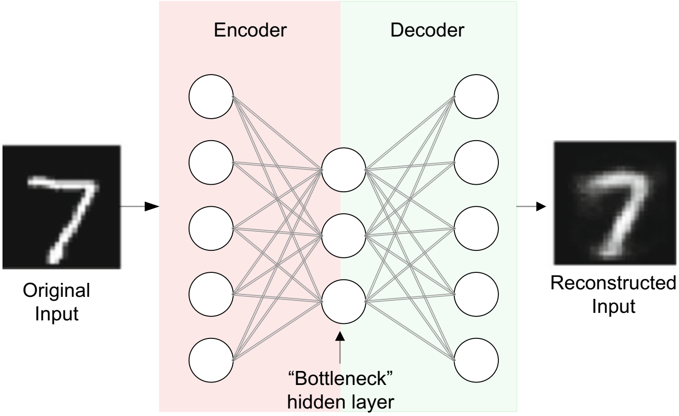


Fig. 4 Architecture of vanilla autoencoder

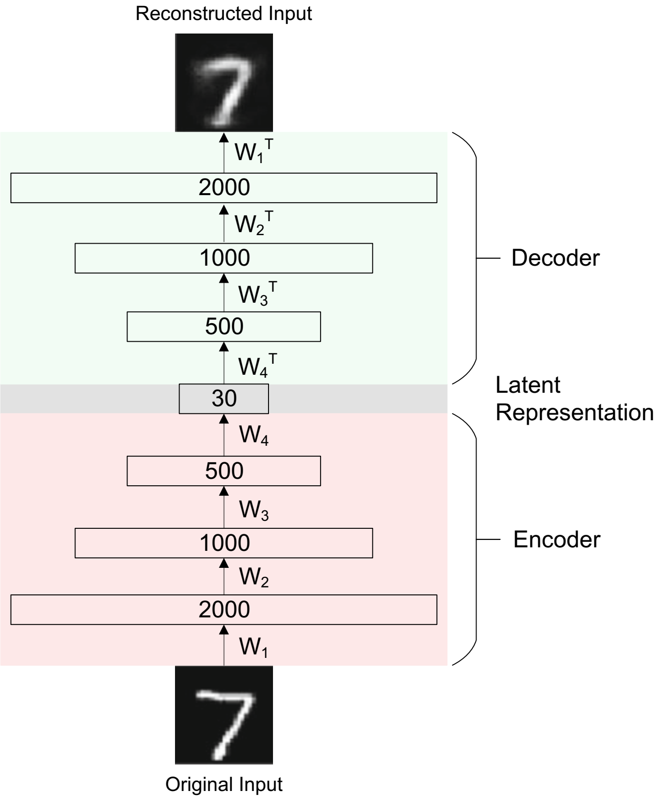


Fig. 5 Deep autoencoder [11]

Divergent autoencoder (DIVA)

Divergent autoencoder [12] is good at solving the N-way classification tasks. It transforms input into distributed representational space. The deviation between reconstructed and original input is used for classification.

Convolutional autoencoder (CONV AE)

Instead of using fully connected layers, convolution operators used in convolutional autoencoder extracts useful representation from input data. In this, input image is downsampled to get the latent representation having lesser dimensions and autoencoder is forced to learn the compressed representation of the image. Figure 6 shows the working of convolutional autoencoder.

Encoder is implemented as a typical convolutional pyramid in which each convolution layer is followed by the max pooling layer for reducing the dimensions of an image. As shown in Fig. 6, the dimensions of grey scale input image are 28 × 28 × 1 (vector with 784 dimensions). By successive application of convolution and max-pooling layers, the reduced latent representation of the image with dimensions

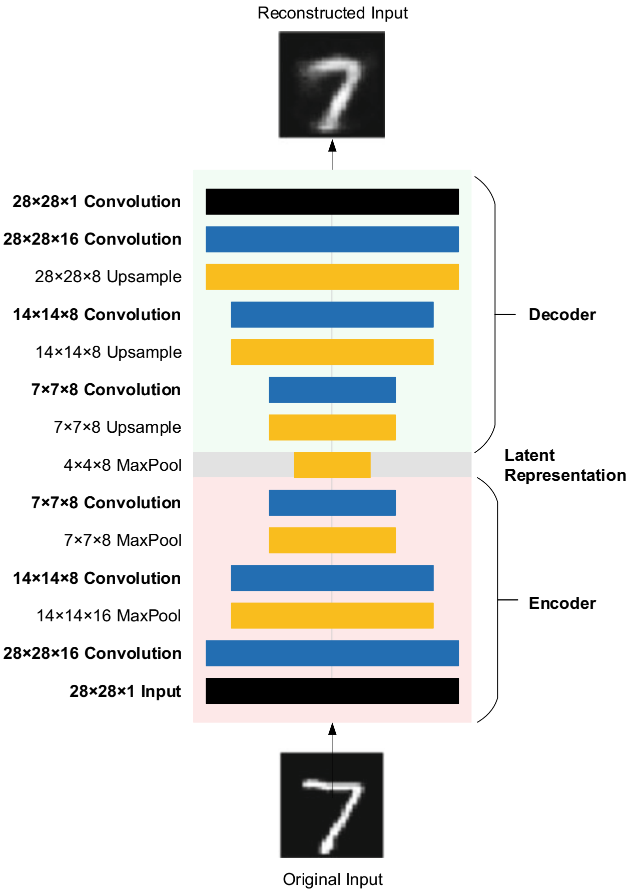


Fig. 6 Architecture of convolutional autoencoder [69]

4 × 4 × 8 is obtained. Decoder converts a narrow representation of image into wide reconstructed image having dimensions 28 × 28 × 1 by successive application of upsampling and transposed convolution. The compression is lossy in convolutional autoencoder. Generally, conversion of narrow representation of image into expanded one can be done using 2 ways viz. upsampling and transposed convolution (deconvolution). Upsampling performs resizing of image by stretching it. The techniques like nearest neighbor interpolation or bilinear interpolation are used for upsampling.

As claimed in [13], nearest neighbor interpolation works better for upsampling. Transposed convolution works exactly as convolution layers, but in reverse manner. For example, convolving a 3 × 3 kernel over image patch of size 3 × 3 would result into patch of one unit in the convolution layer. On the contrary, one unit of patch in the input layer would expand to a patch of 3 × 3 in the transposed convolution. Applying the transposed convolution on images results into generation of checkerboard artifacts on the reconstructed images [13].

For accurately extracting the information from a network, many problems in the domain of computer vison, social network analysis and natural language processing are represented using graph or the network structure. Depth-based subgraph convolutional autoencoder (DS-CAE) [14] models node content information and network structure for network representation learning. It maps graphs to high-dimensional non-linear spaces preserving both the local and global information in the original space. It uses convolution filters for extracting the local features by convolving over the complete set of sub-graphs of a vertex.

RNN based AE

Sequence prediction problems are challenging to handle since the length of input sequences varies in sequence prediction problems and most of the neural networks require fixed length input for processing. Another challenge is temporal ordering of observations make feature extraction a difficult task since providing an input to supervised neural network models need domain expertise. Many applications based on predictive modeling need prediction as output which itself is a sequence. Therefore, recurrent neural networks (RNNs) such as Long Short-Term Memory (LSTM) are designed to support sequence data as input. RNN Encoder-Decoder model proposed in [15] is good at handling the sequence prediction problem like statistical machine translation. The encoder and decoder in this model are built using recurrent neural networks.

In RNN based AE, variable length input sequence is mapped to fixed-length vector using encoder. This fixed-length vector is mapped back to variable-length output sequence using decoder. Both encoder and decoder have been jointly trained for maximizing the probability of output sequence given an input sequence.

LSTM autoencoder

Srivastava et al. [16] described the LSTM autoencoder as an extension to RNN based AE for learning the representation of time series sequential data, audio, text and videos. In this model, encoder and decoder are built using LSTM. Encoder LSTM accepts a sequence of vectors in the form of images or features. Decoder LSTM recreates the target sequence of input vectors in the reverse order. As claimed by the authors, recreating the input sequence in reverse order makes the optimization process more tractable. Decoder is designed using 2 ways viz. conditional and unconditional. A conditional decoder receives previously constructed output frame as input whereas unconditional decoder does not receive previously created output frame.

Composite LSTM autoencoder

Composite LSTM autoencoder [16] performs both the tasks of reconstructing the sequence of video frames and prediction of next video frame. In this, encoder LSTM represents such a state based on which next few frames can be predicted, and input frames can be reconstructed.

3.2 Regularized Autoencoders

Regularized autoencoder (RAE)

Regularized autoencoders use a loss function so that model supports the following properties as ability to reconstruct the input by learning the distribution of data, sparse representation, and robustness to handle noisy data [17]. Though model can serve as a nonlinear and overcomplete autoencoder, it can still learn the salient features from distribution of input data.

Sparse autoencoder (SAE)

Sparse autoencoders are used for extracting the sparse features from the input data. The two ways for imposing the sparsity constraint on the representation can be given as follows. (i) applying penalty on the biases of the hidden units [18, 19] (ii) penalizing the output of latent space (hidden unit) activation [20]. In sparse autoencoders [18, 21] hidden units are more than input units although only a small number of hidden units can be active at some time. Sparse autoencoders impose sparsity penalty $\Omega(h)$ on the hidden layer in addition to the reconstruction error for preventing the output layer from copying input data. Therefore, the loss function can be given as shown in Eq. (2) where $\Omega(h)$ is sparsity penalty.

$$L = |x - g(f(x))| + \Omega(h) \tag{2}$$

A variant of sparse autoencoder, specifically 9-layered locally connected sparse encoder with pooling and local contrast normalization has been put forth in [22]. The model trained using this autoencoder performs face detection using unlabeled data. This autoencoder is invariant to translation, scale and out-of-plane rotation. Much research work has focused on representation learning from the data. Feature representation algorithms based on nonnegativity-constrained autoencoder and SAE causes feature redundancy and overfitting due to duplicate encoding and decoding receptive fields. Cross-variance based regularized autoencoders regularize the feature weight vectors to alleviate feature redundancy and reduce overfitting [23].

Stacked autoencoder

A neural network having multiple layers of sparse autoencoder is known as stacked autoencoder. Adding more hidden layers to an autoencoder causes reduction of high dimensional data. This enables a compressed representation to exhibit the salient features of data such that every ith layer has more compact representation than layer at $i - 1$ level. In stacked autoencoder, each successive layer in model is optimally weighted and it is non-linear.

Saturating autoencoder (SATAE)

Saturating autoencoder [24] constraints the ability of autoencoder to reconstruct the input data which are not near the data manifold. It acts as a latent state regularizer for the autoencoders whose activation functions of latent space possess at least one saturated region with zero-gradient. Sparse and saturating autoencoders regularize their latent states to prevent autoencoder from learning the reconstruction of the input data and thus focuses on improving the expressive power of autoencoders to represent the data-manifold.

Hessian regularized sparse autoencoder (HSAE)

HSAE [25] encompasses three terms viz. reconstruction error, sparsity constraint and Hessian regularization. Reconstruction error computes the loss between input sample and reconstructed sample. Sparsity constraint enables to learn the hidden representation of data and makes the model robust to noise. Hessian regularization preserves the local structure and controls the linearly varying learned encoders along the manifold of data distribution. This autoencoder can be improved to support large scale multimedia data by parallelizing the autoencoder algorithm.

Contractive autoencoder (CAE)

Rifai et al. [26] put forth a contractive autoencoder model with an aim to learn the robust representation of data. Explicit regularizer is added in the objective function of contractive autoencoder to make the model robust to slight variation in input data. Equation (3) [17] showing the loss function of contractive autoencoder is given as

$$L = |x - g(f(x))| + \Omega(h) = |x - g(f(x))| + \lambda \left\| \frac{\partial \mathrm{f(x)}}{\partial \mathrm{x}} \right\|_{\mathrm{F}}^{2} \tag{3}$$

where penalty term $\Omega(h)$ for the hidden layer is calculated w.r.t. input x which is known as Frobenius norm of the Jacobian matrix. Sum of square of all elements is given by Jacobian matrix. Contractive autoencoders are better than denoising autoencoders for feature learning. The mapping generated by penalty term results into strong contraction of data and therefore it is termed as contractive autoencoder.

Higher order CAE

Rifai et al. [27] extended the approach of contractive autoencoder for improving the robustness to corrupted data and stabilizing the learned representation around the training points for manifold learning. They explicitly performed the regularization of latent state representation using first order derivative (Jacobian norm) and second order derivative (Hessian norm) for improving feature learning and optimizing the classification error.

Alain et al. [28] mentioned that both denoising and contractive autoencoders have similar training criterion i.e. denoising autoencoder having small corruption noise can be considered as a variant of contractive autoencoder where contraction is applied on entire reconstruction function instead of just the encoder. Both autoencoders support unsupervised and transfer learning [29].

Zero-bias autoencoder

Whenever contraction penalties or sparse penalties are used as explicit regularization strategies while training on large number of hidden units, hidden biases reach large negative biases. The reason behind this is that hidden layers serve the purpose of both representing the input data and maintaining the sparse representation. Therefore, to avoid the detrimental effects of large valued negative biases, Konda et al. [30] put forth zero-bias autoencoder which acts as an implicit regularizer and allows training the model without explicit regularization by simply minimizing the reconstruction error.

k-sparse autoencoder

The k-sparse autoencoder put forth in [31] is an autoencoder with linear activation in which only the selected k highest neurons from hidden layer are used for reconstructing the input. This autoencoder approximates sparse coding algorithm which uses "iterative thresholding with inversion method" in sparse recovery phase. It enforces sparsity across different channels, known as population sparsity. The population sparsity is exactly enforced in the hidden units and this autoencoder does not need any non-linearity and regularization.

Winner-Take-All Autoencoders (WTA)

Winner-Take-All Autoencoders [32] have been put forth for hierarchical sparse representation of data using unsupervised learning. The first variant of WTA, namely, fully connected WTA (FC-WTA) enforces lifetime sparsity constraint [33] on the hidden unit activations with the help of mini batch statistics. Lifetime sparsity is applied across whole training examples.

Another variant—convolutional WTA (CONV-WTA) combines the benefits of convolutional networks and autoencoders for learning shift-invariant sparse representation of data. Both these variants are scalable to large datasets like ImageNet for classification and support unsupervised feature learning.

Smooth autoencoder

Unlike conventional autoencoders which reconstruct the data from encoding, smooth autoencoder [34] reconstructs the target neighbors of sample by using encoded representation of each sample. This enables to capture similar local features and enhance inter-class similarity for classification task.

Deep kernelized AE

The variant of stacked AE, namely deep kernelized AE [35] leverages user-defined kernel matrix and learns to preserve non-linear similarities in the input space. By this autoencoder, user can explicitly control the notion of similarity in the input data by encoding it in a positive semi-definite kernel matrix. This autoencoder is useful for classification tasks and visualization of high dimensional data.

Graph structured autoencoder

A graph regularized version of autoencoder has been proposed in [36]. The first variant of this AE works in unsupervised manner for image denoising. Another variant, namely, low-rank representation regularized graph autoencoder incorporates subspace clustering terms into its formulation to perform the task of clustering. The third variant of graph structured AE incorporates label consistency for solving single- and multi-label classification problems in supervised settings.

Group sparse autoencoder (GSAE)

Sankaran et al. [37] proposed Group sparse autoencoder based representation learning approach. It works in supervised learning mode and uses $\ell 1$ and $\ell 2$ norms utilizing the class labels for learning the supervised features for the specific task. The optimization function in this AE works on majorization-minimization approach. It performs classification using cost-sensitive version of support vector machine with radial basis function network.

3.3 Robust Autoencoders Tolerant to Noise

Denoising autoencoder

For increasing the robustness of autoencoder to changes in the input, Vincent [38, 39] put forth a denoising autoencoder. Rather than penalizing the loss function for regularization, noise is added to the image x and this noisy image $\tilde{x}$ is fed as input to the denoising autoencoder. Figure 7 gives the general architecture of denoising autoencoder. Stochastic mapping is used for denoising purpose in denoising AE. In DAE, corrupted copy of the input data is created by introducing some noise. It encodes the input and tries to undo the effect of corruption applied to input image.

This autoencoder is trained to generate the cleaned images from the noisy one. As it is harder to generate the cleaned image, this model requires more deep layers and more feature maps. For every iteration of the training, the network computes a loss between reconstructed noisy image obtained from decoder and the original noise-free image and tries to minimize the loss.

Denoising Autoencoder Self-organizing Map (DASOM)

DASOM [40] addresses the issue of integrating the non-linearities of neurons into networks for modelling more complex functions. It works by interposing a layer

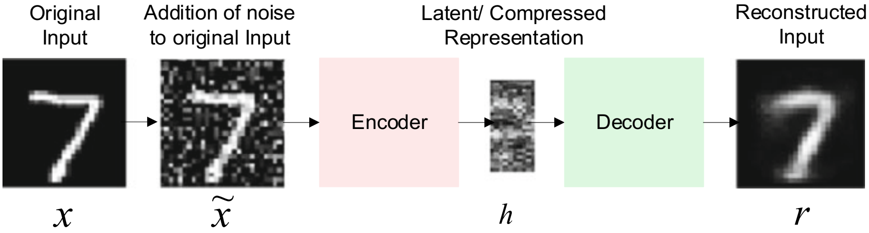

Original Input
Addition of noise to original Input
Latent/ Compressed Representation
Reconstructed Input
Encoder
Decoder
x
$\tilde{x}$
h
r

Fig. 7 Architecture of denoising autoencoder [38]

of hidden representation between the input space and the neural lattice of the self-organizing map. This AE is useful in optical recognition of images and text.

Stacked denoising autoencoder (SDAE)

Stacking DAEs for creating a deep network works in similar manner as stacking the RBMs in deep belief networks [11]. In SDAE, input corruption is applied to each individual layer for initial denoising and training to learn the salient features from data. After learning of latent encoding function f_θ, it is applied on the uncorrupted input onwards. For training the next layers in the model, uncorrupted input from previous layers are used as clean input for the next layer.

Marginalized denoising autoencoders (mDAE)

In DAEs, input data needs to be corrupted many times during training phase and this causes increase in size of training data and incurs more computational resources. This problem even gets worse when dimensionality of input data is very high. Marginalized denoising autoencoders [41] address this issue by approximately marginalizing out the data corruption process during training. It accepts the multiple corrupted copies of input data in every iteration of training and outperforms DAE with few training epochs. Instead of using explicit data corruption process, mDAEs implicitly marginalize out the reconstruction error over possible data corruption from corrupting distribution such as additive Gaussian and Unbiased Mask-out/drop-out.

Hierarchical autoencoder

In stacked DAE, only the final layer is responsible for reconstructing the input sample and intermediate layers do not directly contribute for reconstruction.

Hierarchical autoencoder [42] is designed such that intermediate layers provide complementary information and output of each layer is fused to get final reconstructed sample. This autoencoder is based on asymmetric autoencoder in which a stacked autoencoder has only one decoder. A shallow nature of decoder alleviates the need to train multiple layers, and therefore layers can directly contribute for reconstructing the input.

3.4 Generative Autoencoders

Variational autoencoder (VAE)

In this autoencoder, bottleneck vector (latent vector) is replaced by two vectors, namely, mean vector and standard deviation vector. Variational autoencoders [43] are based on Bayesian inference in which the compressed representation follows probability distribution. Unlike vanilla autoencoders which learn the arbitrary encoding function for obtaining the salient features, variational autoencoders learn the parameters of probability distribution which model the input training data; therefore VAEs are complex in nature. The encoder network is forced to generate the latent vectors

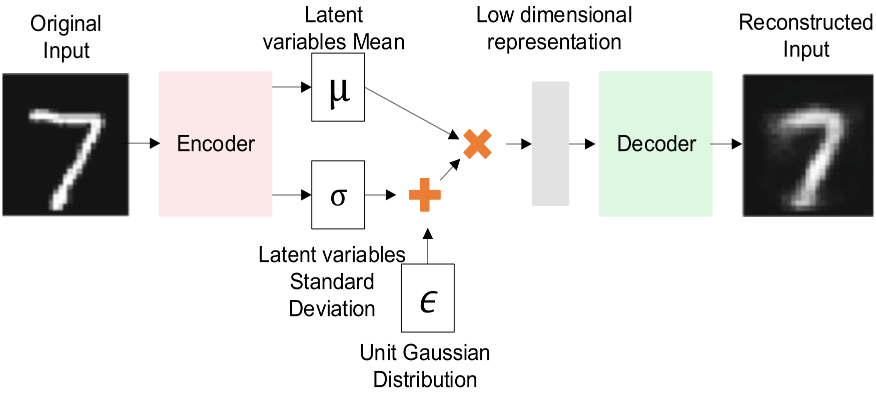

Original Input
Latent variables Mean
Low dimensional representation
Reconstructed Input
Encoder
μ
σ
Decoder
Latent variables Standard Deviation
ϵ
Unit Gaussian Distribution

Fig. 8 Variational autoencoder [43]

following the unit Gaussian distribution. This constraint differentiates VAE from standard autoencoder.

The working of variational autoencoder is depicted in Fig. 8. Generally, there is a tradeoff between how accurately the network reconstructs the images and how closely the latent variables match the unit Gaussian distribution. The reconstruction error (generative loss) is measured using mean squared error whereas Kullback–Leibler (KL) divergence loss measures the closeness of latent variables with unit Gaussian distribution. Typical variational autoencoders are based on strong assumption that posterior distribution is factorial whose parameters can be approximated using non-linear regression based on observed samples.

Importance weighted autoencoder (IWAE)

Importance weighted autoencoder [44] is a generative model and a variant of variational autoencoder. It has similar architecture as that of VAE with the exception that it utilizes strictly tighter log-likelihood lower bound obtained from importance weighting and learns latent representation of data better than VAE.

Adversarial autoencoder

Adversarial autoencoder [45] uses generative network to perform variation inference for both continuous and discrete latent vectors in probabilistic autoencoders. Figure 9 shows adversarial autoencoder which constitutes standard autoencoder and a network for adversarial training. Standard AE reconstructs an image from latent vector *h*. Adversarial training network is used for discriminatively predicting whether samples are emanated either from hidden code or user specified distribution. VAE uses KL divergence loss for imposing prior distribution on hidden code vector of autoencoder, whereas adversarial autoencoder applies adversarial training method to match the aggregated posterior of the hidden code representation with the prior distribution.

Varied distribution in multi-view data causes view discrepancy. It is important in many practical applications to learn the common representations from multi-view data. Wang et al. [46] proposed unsupervised multi-view representation learning

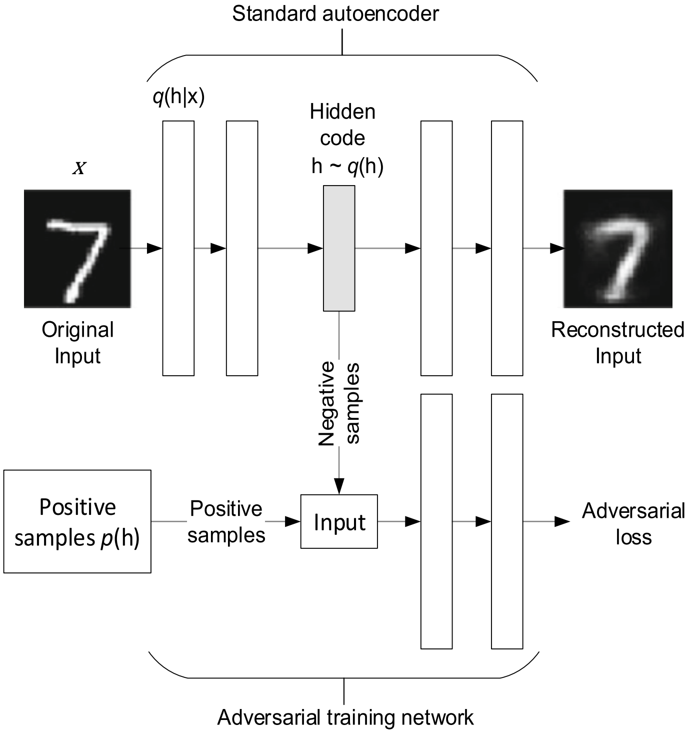


Fig. 9 Adversarial autoencoder [45]

method, namely, adversarial correlated autoencoder which learns common representation from multi-view data.

Wasserstein Autoencoders (WAE)

The drawback of variational autoencoders is that it generates blurry images when natural images are used as input training data. The visual quality of images is quite impressive when generative adversarial networks are used. But generative adversarial networks suffer from "mode collapse" issue when a trained model is unable to learn variability in the true data distribution. Wasserstein autoencoders have been designed to combine the best properties of both VAEs and GANs in a unified way.

WAE [47] is used for designing the generative models based on optimal transport perspective. It penalizes the Wasserstein distance between model distribution and target distribution. In this, encoded training distribution is matched with the prior distribution. Like variational autoencoder, objective function of WAE constitutes a reconstruction cost and a regularizer penalizing the discrepancy between prior distribution and distribution of encoded points.

Adversarially Regularized Autoencoders (ARAE)

Adversarially Regularized Autoencoder [48] is based on WAE [47] and it is an extended version of adversarial autoencoder [45] to support discrete sequences such as discretized images or text sequences. This model allows manipulating the variables in latent space to incorporate change in the output space. It also handles sequential data by incorporating both learned and fixed prior distribution.

Dynencoder

Dynencoder put forth by Yan et al. [49] represents spatiotemporal information of a video. It constitutes three layers. The first layer performs mapping of input x_t to latent state h_t. Next hidden state $\tilde{h}_{t+1}$ is predicted by the second layer using current state h_t. The final layer performs mapping from predicted hidden state $\tilde{h}_{t+1}$ to generate approximated input frame $\tilde{x}_{t+1}$. Initially, each layer is trained separately in pre-training phase. Once pre-training is over, an end-to-end fine-tuning of network is done.

Stacked what-where auto-encoder (SWWAE)

SWWAE [50] synergistically combines the advantages of discriminative and generative models and acts as unified model to support unsupervised, semi-supervised and supervised representation learning without making use of sampling during training. It encodes the input using convolution net [51] and performs reconstruction using deconvolution net [52]. It consists of feedforward convolution network coupled with a feedback deconvolution network. Encoder is composed of convolution layer with ReLU activation followed by max pooling layer. A pooling layer splits the information into "what" and "where" components which are being described as max values and switch positions respectively. The "what" variables inform the next layer regarding the incomplete information about position and "where" variable inform the corresponding feedback decoder the position (location) of salient features. The crux of this model can be stated as whenever a layer is shown via many-to-one mapping, this model computes complementary variables for reconstructing the input.

4 Factors Affecting Overall Performance of Autoencoders

Factors like training procedure, regularization, activation functions, etc. play crucial role in successful implementation of any autoencoder based model.

4.1 Training

All the training procedures of autoencoder must maintain a tradeoff between following two requirements.

1. Learning a latent representation of the input sample such that input can be reconstructed via decoder through approximation: It is expected that autoencoder should reconstruct the input by following the data-generating distribution.
2. Fulfilling the sparsity constraint or regularization penalty: Sparsity constraint enables to limit the capacity of autoencoder. Regularization penalty is required for imbibing special mathematical properties in the learned encodings.

Satisfying above two requirements together is important since they enforce hidden representation to capture the salient features from data based on its distribution. Autoencoder should learn the variations in data so that input data can be reconstructed.

Autoencoders may be considered as a special case of feedforward neural networks, and therefore they can be trained with same techniques as that of feedforward networks such as minibatch gradient descent. SGD [53], and its variants Adam [54], AdaGrad [55], and RMSProp are some algorithms used for optimization of weights and biases in autoencoders during training. Other algorithms include L-BFGS and conjugate gradient [56].

All these algorithms are based on gradient descent technique. Gradient descent algorithm finds the parameters of a function f in the direction of steepest slope and minimizes a cost function. Back-propagation algorithm [57] is used for calculating the gradients of loss function from last layer to a first layer in a neural network to adjust the weights.

Autoencoders may be trained using recirculation training algorithm [58] which compares the activations of network on the training data and the activations on reconstructed data. Generative models implemented via deep architectures generally follow greedy layer-wise pre-training strategy. Another way to train the deep model is to train a stack of the shallow autoencoders. With the growing volume of large scale unlabeled data and need to investigate different types of regularizers, Zhou et al. [59] proposed unsupervised learning method for jointly training all layers of deep autoencoder. In this, single objective for training deep autoencoder encompasses global reconstruction objective having local constraints on hidden layers to enable joint training.

4.2 Objective Function

The objective function of autoencoder encompasses reconstruction error and penalty terms expressed via sparsity constraints and/or regularization.

Reconstruction error can be calculated using mean squared error (MSE), cross entropy and correntropy [60]. MSE gives an average squared difference between the actual and predicted values. Cross-entropy is used for quantifying the difference between two probability distributions. Correntropy checks the equality of probability density of two distributions. Correntropy measure is more robust to the outliers than MSE.

Regularization helps to make the model generalize well on new unseen data. It can be performed via data, network architecture, optimization, error function, and regularization term [61]. The effectiveness of generative models can be improved by discriminative regularization. In this, supervised learning algorithms augment generative models to discriminate which features of data are worth to be represented [62]. To understand how well autoencoders represent the data, energy function [63] or un-normalized score [64] can be used which relate AE to probabilistic model such as RBM. To avoid overfitting of autoencoders, regularization term is added to the objective function causing weight decay [65].

Weight decay improves generalization by choosing the smallest vector to surpass the irrelevant components of the weight vector. Other ways of performing regularization are contraction and sparsity constraints. Encodings generated by basic autoencoders do not possess special properties. To imbibe mathematical properties in these encodings, some regularization methods add penalty function to the objective function. Sparse autoencoder, denoising autoencoder and contractive autoencoder are some popular examples of regularized autoencoders. The penalty terms in regularized autoencoders can be Frobenius norm of the Jacobian (first order derivative) or Hessian norm (higher order derivative).

Regularization penalties may be either applied on activations of hidden units or activation of output layer. Regularizers applying penalty on activations of hidden units are termed as latent state regularizers.

While learning the sparse representation of data, sparsity constraint may be applied across channels on population sample (population sparsity constraint) [31] or across training examples (lifetime sparsity constraint) [33]. Another constraint, namely, spatial sparsity constraint [32] is used for regularizing the autoencoder, and it requires contribution of all dictionary atoms in reconstruction of input data. Rather than reconstructing the input data from all hidden units of the feature maps, spatial sparsity constraint selects single largest hidden unit from each feature map and sets remaining units and their derivatives to zero. This makes sparsity level equal to the number of feature maps. The decoder reconstructs the output with the help of active hidden units and reconstruction error is back-propagated through these active hidden units only.

Autoencoders are exceptionally good at learning the properties/features of data. In order to verify how much useful information is exhibited by the features constructed by the autoencoders for the task of classification, autoencoder node saliency method based on principle of information theory has been proposed in [66]. In this, hidden nodes are ranked according to their relevance to a learning task using supervised node saliency method. Furthermore, interestingness of the latent representation is computed using normalized entropy difference for verifying the classification ability of the highly ranked nodes.

4.3 Activation Functions

Activation function transforms summed weighted input into the activation of node or output. They play an important role of propagating the gradients in the network. Activation function models the nonlinear behavior of most neural networks. Sigmoid function (Standard logistic function) is the most widely used activation functions in autoencoders. Another sigmoid function—hyperbolic tangent function is symmetric over origin. As it generates steeper gradients, it should be preferred over other activation functions [67]. Use of ReLU as an activation function is popular choice in deep learning models. As ReLU function outputs 0 for negative values, it may hamper the performance of autoencoders by degrading the reconstruction process while decoding. Scaled exponential linear units (SELUs) [68] enable to train the deep model having multiple layers and learn robust representation by employing strong regularization. It addresses the issue of vanishing and exploding gradients. Activation functions like linear function, binary function and ReLU are seldom used in AEs.

4.4 Layer Size and Depth

Though autoencoders can be trained with single encoding layer and single decoding layer, it is important to train the autoencoder using deep layers for better representation learning. As previously mentioned, both encoder and decoder in autoencoder can be thought of as feedforward neural network (FNN); hidden layer in FNN can approximate any function having arbitrary accuracy given that hidden layer has enough nodes. This proves that autoencoder having single hidden layer can represent the identity function following the distribution. But, as the mapping from input to latent state (hidden code) is shallow, arbitrary constraints can't be enforced in order to make the hidden code follow sparse representation. A deep autoencoder having at least one hidden layer (with enough hidden nodes) is able to approximate any input data by mapping it to code. Depth of deep architecture reduces the computational cost of function/data representation, and exponentially reduces the quantity of input training data for learning the functions. Deep autoencoders are better at data compression than their shallow or linear counterparts [11].

5 Applications of Autoencoders

Autoencoders are found to be useful in many tasks such as classification, prediction, feature learning, dimensionality reduction, anomaly detection, visualization, semantic hashing, information retrieval, and other domain specific traits. Some autoencoders are designed considering the problem to be solved.

Dimensionality reduction is one of most conventional applications of autoencoders. Lower dimensional representations enable to improve the performance of model on the tasks like classification. The working of variants of autoencoder has been widely investigated for classification task in the literature. Autoencoders like RNN based autoencoder, LSTM encoder are preferably used for solving sequence prediction problems.

Autoencoders such as deep convolutional autoencoders have been used for anomaly detection, feature engineering. In case of anomaly detection, autoencoders are trained for normal training data, and so reconstructed data also follow the distribution of normal data and can't generate the data not seen beforehand (anomalous patterns). Therefore, reconstruction error is treated as an anomaly score. Autoencoders have also been used for semantic hashing to be applied on both textual data and images. Hashing makes the search process faster by encoding the data to binary codes. By and large, autoencoders have been extensively applied in multitude of tasks. Table 1 gives characteristics of different autoencoders along with their applications.

Table 1 Applications of autoencoders

Architecture	Characteristics	Applications
Deep autoencoder [11, 70]	Performs pretraining via stack of RBMs, unrolling the structure, creating a deep autoencoder which can be finetuned using back-propagation	Classification, regression, compression, semantic hashing, geochemical anomaly detection
Convolutional autoencoders [69, 71–73]	Preserves spatial locality by sharing weights in each convolution layer	Reconstruction of missing parts in an image, image colorization, generating super resolution images, anomaly detection, indoor positioning system
DS-CAE [44]	Performs local feature extraction on graphs and networks using convolution operation	Network and graph representation learning
RNN based AE [15]	Handles variable-length input and recreates variable length output for sequence data and applicable to handle sequence-to-sequence prediction problems	Statistical machine translation, image captioning, chat bots, generating commands for gestures in a sequential manner
LSTM AE and Composite LSTM model [16]	Reconstructs input vectors/frames and predicts next vectors/frames and applicable to handle sequence-to-sequence prediction problems	Action recognition, text processing

(continued)

Table 1 (continued)

Architecture	Characteristics	Applications
Sparse autoencoder [18–20, 74]	Applies sparsity penalty on hidden layer to prevent output layer copying input data	Classification, segmentation, inpainting, compression, interpolation methods for super-resolution
Saturating autoencoder [24]	Good for feature extraction, constraints the ability of reconstructing the inputs which are not near the data manifold	Classification, denoising
HSAE [25]	Incorporates reconstruction error, sparsity constraint and Hessian regularization for robust manifold learning	Classification
Zero-bias autoencoder [30]	Acts as implicit regularizer for learning the very high dimensional features intrinsically	Feature learning from high dimensional data such as video and images
k-sparse autoencoder [31]	Enforces sparsity constraint on hidden layer and select *k* neurons for reconstruction	Shallow and deep discriminative learning tasks, unsupervised feature learning
FC-WTA [32]	Applies lifetime sparsity constraint on hidden unit for sparse data representation	Classification, unsupervised feature learning
CONV-WTA [32]	Learns shift-invariant sparse representation of data by applying spatial and lifetime sparsity constraints	Classification, unsupervised feature learning
Smooth AE [34]	Reconstructs target neighbors of each sample by respective encoded representation of sample	Data manifold representation, classification
Denoising autoencoder [38, 39]	Reconstructs the correct data from corrupted or noisy input data, supports unsupervised and transfer learning	Removing watermarks, Image inpainting
mDAE [41]	Considers multiple copies of corrupted data and implicitly marginalizes out the reconstruction error over data corruption	Representation learning
Hierarchical autoencoder [42]	Good at handling analysis and synthesis problems due to shallow nature of decoder	Recommender systems

(continued)

Table 1 (continued)

Architecture	Characteristics	Applications
Variational autoencoder [43, 75, 76]	Generates new data augmenting the sample data and works as a typical generative adversarial model	Generative modeling, missing data imputation, dimensional sentiment analysis
IWAE [44]	Learns richer latent representation than VAEs by utilizing importance weighting	Generative modeling
Adversarial autoencoder [45]	Uses adversarial training method for variation inference	Dimensionality reduction, classification, unsupervised clustering, disentangling the style and content of images
Wasserstein Autoencoders [47]	Minimizes optimal transport cost in generative models	Generative modeling
ARAE [48]	Works as deep latent variable model and produces robust representation for discrete sequence data following both WAEs and adversarial autoencoders	Unaligned style transfer for text (Discrete data)
Dynencoder [49]	Captures video dynamics by spatiotemporal representation of video	Synthesizing dynamic textures from video, classification
SWWAE [50]	Learns factorized representation using convolution and deconvolution net to encode invariance and equivariance properties	Factorized representation learning
Contractive autoencoder [57]	Applies contractive penalty in the activation unit of latent state, captures the variation stated by data, supports unsupervised and transfer learning	Classification
CFAN AE [77]	Cascade of autoencoders used for coarse to fine level processing	Face alignment identification
Stacked Convolutional AE [78]	Stack of convolutional autoencoder trained using online gradient descent	Hierarchical feature extraction, unsupervised learning, classification
Autoencoder for words [79]	Performs encoding of words	Indexing, ranking and categorizing the words

(continued)

Table 1 (continued)

Architecture	Characteristics	Applications
Binary autoencoder [80]	Represents hidden code layer by binary vector, performs reconstruction, follows method of auxiliary coordinates	Semantic hashing for images
ARGA and ARGVA [81]	Adversarial models for representing the graph data on lower dimensional space for graph analytics	Graph clustering, graph visualization, link prediction
AdvCAE [46]	Uses unsupervised learning to obtain common representation for multi-view data by following generative modeling	Cross-view classification and cross-view retrieval
Generative Recursive AE [82]	Learns hierarchical scene structures by grouping scene objects during encoding and scene generation during decoding	Generating diverse 3D indoor scenes at large scale
Stacked convolution AE, WGANs, Siamese network [83, 84]	Learns patch-level representation of subjects using unsupervised feature learning	Outlier detection in medical image processing, pathology image analysis
Optimized Deep Autoencoder + CNN [85]	Performs feature extraction using CNN and learns temporal changes in video streams at real time using deep AE	Online data stream analysis, action recognition
Spectral-spatial stacked autoencoder [86]	Extracts spectral and spatial features from hyperspectral images using stacked AE	HSI analysis, Anomaly detection from hyperspectral images
Class Specific Mean Autoencoder [87]	Uses class information of sample data during training for learning the intra-class similarity and performs feature extraction	Adulthood classification from facial images
Stacked AE [88, 89]	Learns structural features from different stages of the deep learning network	Wind power prediction, tourism demand forecasting
Multilayer Perceptron with Stacked DAE [90]	Captures non-linear relationships, complex interactions and structures embedded in the input data	Prediction of gene expression profiles from genotypes

(continued)

Table 1 (continued)

Architecture	Characteristics	Applications
Stacked Contractive AE [91]	Learns non-linear sub-space from 2D and 3D images, handles illumination changes and complex surface shapes in images	3D face reconstruction
Coherent Averaging Estimation Autoencoder [92]	Models cost function as multi-objective optimization problem encompassing reconstruction, discrimination and sparsity terms	Feature extraction of signals in the domain of brain computer interfaces
RODEO [93]	Utilizes universal function approximation capacity of neural networks and leans the reconstruction (non-linear inversion) process from training data	Compressed sensing based real-time MRI and CT reconstruction
Multimodal Stacked Contractive AE [94]	Preservers intra-modality and inter-modality semantic relations in consecutive stages of AE	Multi-modal video classification
Deep AE + DAE [95]	Synergistically combines Deep AE based discriminant bottleneck feature DAE based dereverberation	Distant-talking speaker identification
Distributed deep CONV AE [96]	Learns complex hierarchical structure of big data and leverages processing power of GPUs in a distributed environment	Analysis of large neuroimaging datasets
Deep kernelized AE [35]	Preserves non-linear similarities in the input space by leveraging user-defined kernel matrix	Classification, visualization of high dimensional data
Graph structured autoencoder [36]	Different variants of graph structured AE, each following either supervised learning or unsupervised one	Image denoising, clustering, single- and multi-label classification
Group sparse autoencoder [37]	Applies $\ell 1$ and $\ell 2$ norms for supervised feature learning	Classification, latent fingerprint recognition required in forensic and law enforcement applications
DASOM [40]	Models complex functions by integrating non-linearities of neurons	Optical recognition of images and text

(continued)

Table 1 (continued)

Architecture	Characteristics	Applications
Convolutional cross AE [97]	Cross AE handles cross-modality elements from social media data and CNN handles time sequence	Cross-media analysis
NGBAE [98]	Models semantic distribution of bilingual text through explicitly induced latent variable	Cross-lingual natural language processing applications like bilingual word embeddings
Model-coupled AE [99]	Combines echo state network with the autoencoder	Visualization of time series data, real-valued sequences and binary sequences
Purifying VAE [100]	Projects an adversarial example on the manifold of each class, and determines the closest projection as a purified sample	Defense mechanism for purifying adversarial attacks applicable in surveillance systems

6 Conclusion

Representation learning from data plays a crucial role for successful implementation of deep learning models and helps to perform better generalization and achieve acceptable performance. Autoencoders designed using neural networks work in an excellent way for representation learning from data. The proliferation of deep learning has resulted into wide use of autoencoders due to their inherent feature learning and dimensionality reduction characteristics.

The major contribution of this chapter can be stated as follows. This chapter gives the foundational background of autoencoders and state-of-the-art variants of autoencoder architectures. The graphical taxonomy of autoencoders based on various factors required for designing the autoencoders has been proposed. This chapter sheds light upon role of activation functions, depth and layer size of neural network, training strategies and regularization methods for autoencoders. The summarized overview of autoencoders based on characteristics and applications has also been tabulated in this chapter.

Appendix

List of abbreviations used in this chapter are mentioned in Table 2.

Table 2 List of abbreviations

Abbreviation	Meaning
AdaGrad	Adaptive Gradient
Adam	Adaptive Moment Estimation
AdvCAE	Adversarial Correlated Autoencoder
AE	Autoencoder
ARAE	Adversarially Regularized Autoencoder
ARGA	Adversarially Regularized Graph Autoencoder
ARGVA	Adversarially Regularized Variational Graph Autoencoder
CAE	Contractive Autoencoder
CONV AE	Convolutional Autoencoder
CONV-WTA	Convolutional Winner-Take-All Autoencoder
CT	Computed Tomography
DASOM	Denoising Autoencoder Self-Organizing Map
DIVA	Divergent Autoencoder
FC-WTA	Fully Connected Winner-Take-All Autoencoder
FNN	Feedforward Neural Network
GAN	Generative Adversarial Network
GSAE	Group Sparse Autoencoder
HSAE	Hessian Regularized Sparse Autoencoder
HSI	Hyperspectral Image
IWAE	Importance Weighted Autoencoder
KL	Kullback–Leibler
LDA	Latent Dirichlet Allocation
LSTM	Long Short-Term Memory
LZW	Lempel–Ziv–Welch
mDAE	Marginalized Denoising Autoencoder
MRI	Magnetic Resonance Imaging
MSE	Mean Squared Error
NGBAE	Neural Generative Bilingual Autoencoder
PCA	Principal Component Analysis
RAE	Regularized Autoencoder
RBM	Restricted Boltzmann Machine
ReLUs	Rectified Linear Units
RNN	Recurrent Neural network
SAE	Sparse Autoencoder
SATAE	Saturating Autoencoder

(continued)

Table 2 (continued)

Abbreviation	Meaning
SDAE	Stacked Denoising Autoencoder
SELUs	Scaled Exponential Linear Units
SGD	Stochastic Gradient Descent
SWWAE	Stacked What-Where Autoencoder
VAE	Variational Autoencoder
WAE	Wasserstein Autoencoder
WTA	Winner-Take-All Autoencoder

References

1. Bengio, Y., Courville, A., Vincent, P.: Representation learning: a review and new perspectives. IEEE Trans. Pattern Anal. Mach. Intell. **35**, 1798–1828 (2013)
2. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
3. Pathak, A.R., Pandey, M., Rautaray, S.: Adaptive framework for deep learning based dynamic and temporal topic modeling from big data. Recent Pat. Eng. **13**, 1 (2019). https://doi.org/10.2174/1872212113666190329234812
4. Pathak, A.R., Pandey, M., Rautaray, S.: Adaptive model for dynamic and temporal topic modeling from big data using deep learning architecture. Int. J. Intell. Syst. Appl. **11**(6), 13–27 (MECS-Press)
5. Pathak, A.R., Pandey, M., Rautaray, S., Pawar, K.: Assessment of object detection using deep convolutional neural networks. In: Bhalla, S., Bhateja, V., Chandavale, A.A., Hiwale, A.S., Satapathy, S.C. (eds.) Intelligent Computing and Information and Communication, pp. 457–466. Springer Singapore (2018)
6. Pathak, A.R., Pandey, M., Rautaray, S.: Deep learning approaches for detecting objects from images: a review. In: Pattnaik, P.K., Rautaray, S.S., Das, H., Nayak, J. (eds.) Progress in Computing, Analytics and Networking, pp. 491–499. Springer Singapore (2018)
7. Pathak, A.R., Pandey, M., Rautaray, S.: Application of deep learning for object detection. Procedia Comput. Sci. **132**, 1706–1717 (2018)
8. Pawar, K., Attar, V.: Deep learning approaches for video-based anomalous activity detection. World Wide Web **22**, 571–601 (2019)
9. Pawar, K., Attar, V.: Deep Learning approach for detection of anomalous activities from surveillance videos. In: CCIS. Springer (2019, in Press)
10. Khare, K., Darekar, O., Gupta, P., Attar, V.Z.: Short term stock price prediction using deep learning. In: 2nd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), pp. 482–486 (2017)
11. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science **313**, 504–507 (2006)
12. Kurtz, K.J.: The divergent autoencoder (DIVA) model of category learning. Psychon. Bull. Rev. **14**, 560–576 (2007)
13. Odena, A., Dumoulin, V., Olah, C.: Deconvolution and checkerboard artifacts. Distill (2016). https://doi.org/10.23915/distill.00003
14. Zhang, Z., et al: Depth-based subgraph convolutional auto-encoder for network representation learning. Pattern Recognit. (2019)
15. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation (2014). http://arxiv.org/abs/1406.1078
16. Srivastava, N., Mansimov, E., Salakhudinov, R.: Unsupervised learning of video representations using LSTMs. In: International Conference on Machine Learning, pp. 843–852 (2015)

17. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
18. Poultney, C., Chopra, S., Cun, Y.L., et al.: Efficient learning of sparse representations with an energy-based model. In: Advances in Neural Information Processing Systems, pp. 1137–1144 (2007)
19. Lee, H., Ekanadham, C., Ng, A.Y.: Sparse deep belief net model for visual area V2. In: Advances in Neural Information Processing Systems, pp. 873–880 (2008)
20. Zou, W.Y., Ng, A.Y., Yu, K.: Unsupervised learning of visual invariance with temporal coherence. In: NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning, vol. 3 (2011)
21. Jiang, X., Zhang, Y., Zhang, W., Xiao, X.: A novel sparse auto-encoder for deep unsupervised learning. In 2013 Sixth International Conference on Advanced Computational Intelligence (ICACI), pp. 256–261 (2013)
22. Le, Q.V., et al.: Building high-level features using large scale unsupervised learning (2011). http://arxiv.org/abs/1112.6209
23. Chen, J., et al.: Cross-covariance regularized autoencoders for nonredundant sparse feature representation. Neurocomputing **316**, 49–58 (2018)
24. Goroshin, R., LeCun, Y.: Saturating auto-encoders (2013). http://arxiv.org/abs/1301.3577
25. Liu, W., Ma, T., Tao, D., You, J.H.S.A.E.: A Hessian regularized sparse auto-encoders. Neurocomputing **187**, 59–65 (2016)
26. Rifai, S., Vincent, P., Muller, X., Glorot, X., Bengio, Y.: Contractive auto-encoders: explicit invariance during feature extraction. In: Proceedings of the 28th International Conference on International Conference on Machine Learning, pp. 833–840 (2011)
27. Rifai, S., et al.: Higher order contractive auto-encoder. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 645–660 (2011)
28. Alain, G., Bengio, Y.: What regularized auto-encoders learn from the data-generating distribution. J. Mach. Learn. Res. **15**, 3563–3593 (2014)
29. Mesnil, G., et al.: Unsupervised and transfer learning challenge: a deep learning approach. In: Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop, vol. 27, pp. 97–111 (2011)
30. Konda, K., Memisevic, R., Krueger, D.: Zero-bias autoencoders and the benefits of co-adapting features (2014). http://arxiv.org/abs/1402.3337
31. Makhzani, A., Frey, B.: K-sparse autoencoders (2013). http://arxiv.org/abs/1312.5663
32. Makhzani, A., Frey, B.J.: Winner-take-all autoencoders. In: Advances in Neural Information Processing Systems, pp. 2791–2799 (2015)
33. Ng, A.: Sparse Autoencoder. CS294A Lecture Notes, vol. 72, pp. 1–19 (2011)
34. Liang, K., Chang, H., Cui, Z., Shan, S., Chen, X.: Representation learning with smooth autoencoder. In: Asian Conference on Computer Vision, pp. 72–86 (2014)
35. Kampffmeyer, M., Løkse, S., Bianchi, F.M., Jenssen, R., Livi, L.: The deep kernelized autoencoder. Appl. Soft Comput. **71**, 816–825 (2018)
36. Majumdar, A.: Graph structured autoencoder. Neural Netw. **106**, 271–280 (2018)
37. Sankaran, A., Vatsa, M., Singh, R., Majumdar, A.: Group sparse autoencoder. Image Vis. Comput. **60**, 64–74 (2017)
38. Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.-A.: Extracting and composing robust features with denoising autoencoders. In: Proceedings of the 25th International Conference on Machine Learning, pp. 1096–1103 (2008)
39. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., Manzagol, P.-A.: Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion. J. Mach. Learn. Res. **11**, 3371–3408 (2010)
40. Ferles, C., Papanikolaou, Y., Naidoo, K.J.: Denoising autoencoder self-organizing map (DASOM). Neural Netw. **105**, 112–131 (2018)
41. Chen, M., Weinberger, K., Sha, F., Bengio, Y.: Marginalized denoising auto-encoders for nonlinear representations. In: International Conference on Machine Learning, pp. 1476–1484 (2014)

42. Maheshwari, S., Majumdar, A.: Hierarchical autoencoder for collaborative filtering. In: 2018 International Joint Conference on Neural Networks (IJCNN), pp. 1–7 (2018)
43. Kingma, D.P., Welling, M.: Auto-encoding variational bayes (2013). http://arxiv.org/abs/1312.6114
44. Burda, Y., Grosse, R., Salakhutdinov, R.: Importance weighted autoencoders (2015). http://arxiv.org/abs/1509.00519
45. Makhzani, A., Shlens, J., Jaitly, N., Goodfellow, I., Frey, B.: Adversarial autoencoders (2015). http://arxiv.org/abs/1511.05644
46. Wang, X., Peng, D., Hu, P., Sang, Y.: Adversarial correlated autoencoder for unsupervised multi-view representation learning. Knowl. Based Syst. (2019)
47. Tolstikhin, I., Bousquet, O., Gelly, S., Schoelkopf, B.: Wasserstein auto-encoders (2017). http://arxiv.org/abs/1711.01558
48. Kim, Y., Zhang, K., Rush, A.M., LeCun, Y., et al.: Adversarially regularized autoencoders (2017). http://arxiv.org/abs/1706.04223
49. Yan, X., Chang, H., Shan, S., Chen, X.: Modeling video dynamics with deep dynencoder. In: European Conference on Computer Vision, pp. 215–230 (2014)
50. Zhao, J., Mathieu, M., Goroshin, R., Lecun, Y.: Stacked what-where auto-encoders (2015). http://arxiv.org/abs/1506.02351
51. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al.: Gradient-based learning applied to document recognition. Proc. IEEE **86**, 2278–2324 (1998)
52. Zeiler, M.D., Krishnan, D., Taylor, G.W., Fergus, R.: Deconvolutional networks. In: Conference on Computer Vision and Pattern Recognition, pp. 2528–2535. IEEE (2010)
53. Robbins, H., Monro, S.: A stochastic approximation method. Ann. Math. Stat., 400–407 (1951)
54. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization (2014). http://arxiv.org/abs/1412.6980
55. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. J. Mach. Learn. Res. **12**, 2121–2159 (2011)
56. Le, Q.V., et al.: On optimization methods for deep learning. In: Proceedings of the 28th International Conference on International Conference on Machine Learning, pp. 265–272 (2011)
57. Rumelhart, D.E., Hinton, G.E., Williams, R.J., et al.: Learning representations by back-propagating errors. Cogn. Model. **5**, 1 (1988)
58. Hinton, G.E., McClelland, J.L.: Learning representations by recirculation. In: Neural Information Processing Systems, pp. 358–366 (1988)
59. Zhou, Y., Arpit, D., Nwogu, I., Govindaraju, V.: Is joint training better for deep auto-encoders? (2014). http://arxiv.org/abs/1405.1380
60. Qi, Y., Wang, Y., Zheng, X., Wu, Z.: Robust feature learning by stacked autoencoder with maximum correntropy criterion. In: 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 6716–6720 (2014)
61. Kukačka, J., Golkov, V., Cremers, D.: Regularization for deep learning: a taxonomy (2017). http://arxiv.org/abs/1710.10686
62. Lamb, A., Dumoulin, V., Courville, A.: Discriminative regularization for generative models (2016). http://arxiv.org/abs/1602.03220
63. Kamyshanska, H., Memisevic, R.: The potential energy of an autoencoder. IEEE Trans. Pattern Anal. Mach. Intell. **37**, 1261–1273 (2015)
64. Kamyshanska, H., Memisevic, R.: On autoencoder scoring. In: International Conference on Machine Learning, pp. 720–728 (2013)
65. Krogh, A., Hertz, J.A.: A simple weight decay can improve generalization. In: Advances in Neural Information Processing Systems, pp. 950–957 (1992)
66. Fan, Y.J.: Autoencoder node saliency: selecting relevant latent representations. Pattern Recognit. **88**, 643–653 (2019)
67. LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient backprop. In: Neural Networks: Tricks of the Trade, pp 9–48. Springer (2012)

68. Klambauer, G., Unterthiner, T., Mayr, A., Hochreiter, S.: Self-normalizing neural networks. In: Advances in Neural Information Processing Systems, pp. 971–980 (2017)
69. Leonard, M.: Deep Learning Nanodegree Foundation Course. Lecture Notes in Autoencoders. Udacity (2018)
70. Xiong, Y., Zuo, R.: Recognition of geochemical anomalies using a deep autoencoder network. Comput. Geosci. **86**, 75–82 (2016)
71. Leng, B., Guo, S., Zhang, X., Xiong, Z.: 3D object retrieval with stacked local convolutional autoencoder. Sig. Process. **112**, 119–128 (2015)
72. Ribeiro, M., Lazzaretti, A.E., Lopes, H.S.: A study of deep convolutional auto-encoders for anomaly detection in videos. Pattern Recognit. Lett. **105**, 13–22 (2018)
73. Li, L., Li, X., Yang, Y., Dong, J.: Indoor tracking trajectory data similarity analysis with a deep convolutional autoencoder. Sustain. Cities Soc. **45**, 588–595 (2019)
74. Wan, X., Zhao, C., Wang, Y., Liu, W.: Stacked sparse autoencoder in hyperspectral data classification using spectral-spatial, higher order statistics and multifractal spectrum features. Infrared Phys. Technol. **86**, 77–89 (2017)
75. McCoy, J.T., Kroon, S., Auret, L.: Variational autoencoders for missing data imputation with application to a simulated milling circuit. IFAC PapersOnLine **51**, 141–146 (2018)
76. Wu, C., et al.: Semi-supervised dimensional sentiment analysis with variational autoencoder. Knowl. Based Syst. **165**, 30–39 (2019)
77. Zhang, J., Shan, S., Kan, M., Chen, X.: Coarse-to-fine auto-encoder networks (CFAN) for real-time face alignment. In: European Conference on Computer Vision, pp. 1–16 (2014)
78. Masci, J., Meier, U., Cirecsan, D., Schmidhuber, J.: Stacked convolutional auto-encoders for hierarchical feature extraction. In: International Conference on Artificial Neural Networks, pp. 52–59 (2011)
79. Liou, C.-Y., Cheng, W.-C., Liou, J.-W., Liou, D.-R.: Autoencoder for words. Neurocomputing **139**, 84–96 (2014)
80. Carreira-Perpinan, M.A., Raziperchikolaei, R.: Hashing with binary autoencoders. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015)
81. Pan, S., et al.: Adversarially regularized graph autoencoder for graph embedding (2018). http://arxiv.org/abs/1802.04407
82. Li, M., et al.: GRAINS: generative recursive autoencoders for INdoor scenes. ACM Trans. Graph. **38**, 12:1–12:16 (2019)
83. Alaverdyan, Z., Chai, J., Lartizien, C.: Unsupervised feature learning for outlier detection with stacked convolutional autoencoders, siamese networks and wasserstein autoencoders: application to epilepsy detection. In: Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support, pp. 210–217. Springer (2018)
84. Hou, L., et al.: Sparse autoencoder for unsupervised nucleus detection and representation in histopathology images. Pattern Recognit. **86**, 188–200 (2019)
85. Ullah, A., Muhammad, K., Haq, I.U., Baik, S.W.: Action recognition using optimized deep autoencoder and CNN for surveillance data streams of non-stationary environments. Futur. Gener. Comput. Syst. (2019)
86. Zhao, C., Zhang, L.: Spectral-spatial stacked autoencoders based on low-rank and sparse matrix decomposition for hyperspectral anomaly detection. Infrared Phys. Technol. **92**, 166–176 (2018)
87. Singh, M., Nagpal, S., Vatsa, M., Singh, R.: Are you eligible? Predicting adulthood from face images via class specific mean autoencoder. Pattern Recognit. Lett. **119**, 121–130 (2019)
88. Tasnim, S., Rahman, A., Oo, A.M.T., Haque, M.E.: Autoencoder for wind power prediction. Renewables Wind. Water Sol. **4**, 6 (2017)
89. Lv, S.-X., Peng, L., Wang, L.: Stacked autoencoder with echo-state regression for tourism demand forecasting using search query data. Appl. Soft Comput. **73**, 119–133 (2018)
90. Xie, R., Wen, J., Quitadamo, A., Cheng, J., Shi, X.: A deep auto-encoder model for gene expression prediction. BMC Genom. **18**, 845 (2017)
91. Zhang, J., Li, K., Liang, Y., Li, N.: Learning 3D faces from 2D images via stacked contractive autoencoder. Neurocomputing **257**, 67–78 (2017)

92. Gareis, I.E., Vignolo, L.D., Spies, R.D., Rufiner, H.L.: Coherent averaging estimation autoencoders applied to evoked potentials processing. Neurocomputing **240**, 47–58 (2017)
93. Mehta, J., Majumdar, A.: RODEO: robust DE-aliasing autoencoder for real-time medical image reconstruction. Pattern Recognit. **63**, 499–510 (2017)
94. Liu, Y., Feng, X., Zhou, Z.: Multimodal video classification with stacked contractive autoencoders. Sig. Process. **120**, 761–766 (2016)
95. Zhang, Z., et al.: Deep neural network-based bottleneck feature and denoising autoencoder-based dereverberation for distant-talking speaker identification. EURASIP J. Audio Speech Music Process. **2015**, 12 (2015)
96. Makkie, M., Huang, H., Zhao, Y., Vasilakos, A.V., Liu, T.: Fast and scalable distributed deep convolutional autoencoder for fMRI big data analytics. Neurocomputing **325**, 20–30 (2019)
97. Guo, Q., et al.: Learning robust uniform features for cross-media social data by using cross autoencoders. Knowl. Based Syst. **102**, 64–75 (2016)
98. Su, J., et al.: A neural generative autoencoder for bilingual word embeddings. Inf. Sci. (Ny) **424**, 287–300 (2018)
99. Gianniotis, N., Kügler, S.D., Tino, P., Polsterer, K.L.: Model-coupled autoencoder for time series visualization. Neurocomputing **192**, 139–146 (2016)
100. Hwang, U., Park, J., Jang, H., Yoon, S., Cho, N.I.: PuVAE: a variational autoencoder to purify adversarial examples (2019). http://arxiv.org/abs/1903.00585

The Encoder-Decoder Framework and Its Applications

Ahmad Asadi and Reza Safabakhsh

Abstract The neural encoder-decoder framework has advanced the state-of-the-art in machine translation significantly. Many researchers in recent years have employed the encoder-decoder based models to solve sophisticated tasks such as image/video captioning, textual/visual question answering, and text summarization. In this work we study the baseline encoder-decoder framework in machine translation and take a brief look at the encoder structures proposed to cope with the difficulties of feature extraction. Furthermore, an empirical study of solutions to enable decoders to generate richer fine-grained output sentences is provided. Finally, the attention mechanism which is a technique to cope with long-term dependencies and to improve the encoder-decoder performance on sophisticated tasks is studied.

Keywords Encoder-decoder framework · Machine translation · Image captioning · Video caption generation · Question answering · Long-term dependencies · Attention mechanism

1 Introduction

The solution to a considerable number of the problems that we need to solve falls into the category of encoder-decoder based methods. We may wish to design exceedingly complex networks to face sophisticated challenges like automatically describing an arbitrary image or translating a sentence from one language to another. The neural encoder-decoder framework has recently been exploited to solve a wide variety of challenges in natural language processing, computer vision, speech processing, and even interdisciplinary problems. Some examples of problems that can be addressed by the encoder-decoder based models are machine translation, automatic image and

A. Asadi · R. Safabakhsh (✉)
Computer Engineering and Information Technology Department, Amirkabir University of Technology, Tehran, Iran
e-mail: safa@aut.ac.ir

A. Asadi
e-mail: ahmad.asadi@aut.ac.ir

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_5

video caption generation, textual and visual question answering, and audio to text conversion.

The encoder part in this model is a neural structure that maps raw inputs to a feature space and passes the extracted feature vector to the decoder. The decoder is another neural structure that processes the extracted feature vector to make decisions or generate appropriate output for the problem.

A wide variety of encoders are proposed to encode different types of inputs. Convolutional neural networks (CNNs) are typically used in encoding image and video inputs. Recurrent neural networks (RNNs) are widely used as encoders where the input is a sequence of structured data or sentence. In addition, more complex structures of different neural networks have been used to model complexities in inputs. Hierarchical CNN-RNN structures are examples of neural combinations which are widely used to represent temporal dependencies in videos which are used in video description generation.

Another potential issue with this baseline encoder–decoder approach is that the encoder has to compress all the necessary information of the input into a fixed-size tensor. This may make it difficult for the neural network to model temporal dependencies at both the input and the output. Attention mechanism is introduced to overcome the problem of fixed-length feature extraction as an extension to the encoder–decoder model. The distinguishing feature of this approach from the baseline encoder–decoder is that it does not attempt to encode a whole input into a single fixed-size tensor. Instead, it encodes the input into a sequence of annotation vectors and selects a combination of these vectors adaptively, while decoding and generating the output in each step.

Some of the tasks in which the encoder-decoder model is used to solve the problem are as follows.

1.1 Machine Translation

“Machine translation” (MT) is the task of generating a sentence in a destination language which has the same meaning as the given sentence from a source language. Two different approaches exist in machine translation.

The first approach, called “statistical machine translation” (SMT), is characterized by the use of statistical machine learning techniques in order to automatically translate the sentence from the source language to the destination language. In less than two decades SMT has come to dominate academic machine translation research [1].

The second approach is called “Neural Machine Translation” (NMT). In this category, the encoder-decoder framework was first proposed by Cho et al. [2] in 2014. In the model proposed by Cho et al. [2] a neural network is used to extract features from the input sentence and another neural network is used to generate a sentence word by word from the destination language using the extracted feature vector.

In the neural structures used in NMT, a neural network is trained to map the input sequence (the input sentence as a sequence of words) to the output sequence. This kind of learning is known as "Sequence to Sequence Learning".

Evaluations on the early models of NMT showed that although the generated translations are correct, the model faces extreme problems when translating long sentences [3]. The problem of modeling "long-term dependencies" is one of the most important challenges in the encoder-decoder models. We will drill into that and take a look at the proposed solutions, later in this chapter.

1.2 Image/Video Captioning

Image captioning and video captioning are the problems of associating a textual description to a given image or video which holistically describes the objects and events presented in the input. A wide variety of approaches have been proposed to solve these problems, including probabilistic graphical models (PGMs) and neural encoder-decoder based models.

Encoder-decoder based models for image captioning use a CNN as an encoder to extract a feature vector from the input image and pass it to an RNN as the decoder to generate the caption. The model architecture in this task is the same as that of machine translation except that the encoder uses a CNN to encode the image rather than an RNN.

In video captioning, also called "video description generation", a similar model based on the encoder-decoder architecture is employed to generate a caption for the input video. In video captioning models, the encoder typically consists of CNNs or combination of CNNs and RNNs to encode the input video and the decoder is the same as the decoder in machine translation and image captioning.

1.3 Textual/Visual Question Answering

Textual and visual question answering are the problems of generating an answer to a given question about an article and about an input image, respectively. Models proposed to solve these problems are supposed to generate a short or long answer, given an article or an image, and a question about it as the input. The base model architecture is then similar to that of machine translation, except that the encoder is required to extract a feature vector for a pair of inputs. The decoder is the same as the decoder in machine translation and image/video captioning because it is supposed to generate a sentence describing the meaning of the feature vector generated by the encoder.

1.4 Text Summarization

Proposed models for summarizing a text are supposed to generate a textual summary for the input text. The only constraint on the output is that it is required to describe the same meaning as the input text and its length should be shorter than that of the input. The base architecture of these models is the same as the architecture proposed in machine translation, except that the generated output here is from the same language as the input.

It can be easily seen that the baseline architecture proposed in machine translation is also used in other tasks with minor changes. In addition, the decoders of the models in different tasks are similar since most of them are used to generate a sentence word by word to describe the meaning of the input represented by the feature vector. On the other hand, a wide variety of encoders are used in order to extract appropriate feature vectors depending on the input types in different tasks.

The next section of this chapter discusses the baseline encoder-decoder model. The first encoder-decoder based model proposed in machine translation is introduced in Sect. 3. Section 4, discusses different types of encoders and their applications in details and makes a general perspective of the encoder structures in different problems. Section 5, provides a comprehensive study of the decoder structures, techniques of making deeper decoders, along with their applications in image/video caption generation. Section 6, introduces the attention mechanism and its usage in machine translation, Followed by an empirical study of the attention mechanism in other problems.

2 Baseline Encoder-Decoder Model

In this section, we introduce the very baseline encoder-decoder model. To give a clear picture of the idea, the basic structure for solving the machine translation task is presented in which the model is designed to translate a sentence from a source language to a destination one.

2.1 Background

A wide range of problems in natural language processing, computer vision, speech recognition, and some multidisciplinary problems are solved by encoder-decoder based models. More specifically, some sophisticated problems in which generating an often-sequential output such as text is desired can be solved by models based on the encoder-decoder structure.

The main idea behind the framework is that the process of generating output can be divided into two subprocesses as follows:

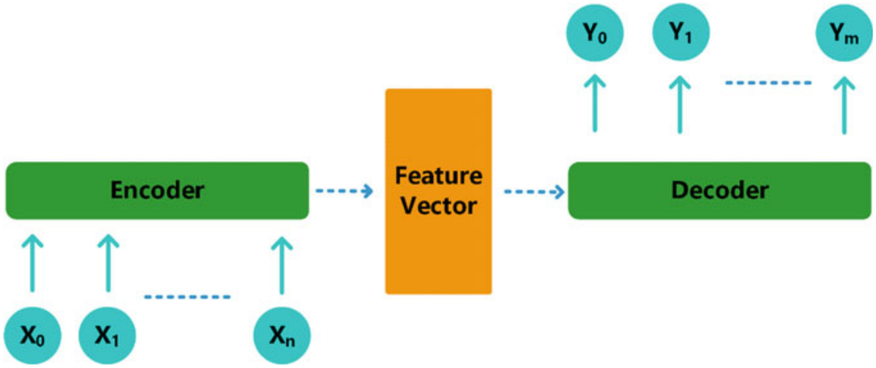


Fig. 1 The basic scheme of the encoder-decoder model

Encoding phase: A given input is first projected into another space by a projection function, called "encoder", in order to provide a "good representation" of the input. The encoder can also be viewed as a feature extractor from the input and the projection process can be expressed by a feature extraction process.

Decoding phase: After the encoding phase, a "latent vector" is generated for the given input that well represents its meaning. In the second phase, another projection function, called "decoder", is required to map the latent vector to the output space.

Figure 1 demonstrates the basic schema of the encoder-decoder framework. Let $\mathrm{X} = \{\mathrm{X}_0, \mathrm{X}_1, \ldots, \mathrm{X}_\mathrm{n}\}$ denote the inputs and $\mathrm{Y} = \{\mathrm{Y}_0, \mathrm{Y}_1, \ldots, \mathrm{Y}_\mathrm{m}\}$ denote the outputs of the problem. The decoder extracts a feature vector from the input and passes it to the decoder. The decoder then generates the output based on the features extracted by the encoder.

2.2 The Encoder-Decoder Model for Machine Translation

Machine translation is the problem in which the encoder-decoder based models were originated and proposed first. The basic concepts of these models are shaped and presented in the machine translation literature. In this section we introduce the basic encoder-decoder structure proposed for machine translation by Cho et al. [2] to shed light on the model and its basics.

2.3 Formulation

Both the input and the output of machine translation models are sentences which can be formulated as a sequence of words. Let $\mathrm{X} = \{\mathrm{X}_0, \mathrm{X}_1, \ldots, \mathrm{X}_{\mathrm{Li}}\}$ denote the input sentence, where x_i is the ith word in it, assuming that the input sentence has L_i

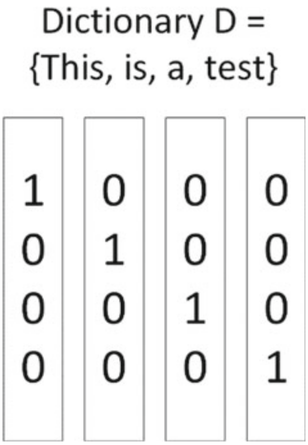


Fig. 2 One-hot vector for each word in a sample dictionary. Sentences can also be modeled using Bag of Words (BoW) technique in which the presence of a word in the sentence is considered without any information about the order of words

words. Similarly, the output sentence could be formulated as $Y = \{y_0, y_1, \ldots, y_{Lo}\}$ in which y_i is the ith word in the output sentence assuming that it has L_o words. Furthermore, all of X_is and y_is are one-hot vectors created from a dictionary of all words in the input and the output datasets.

A one-hot vector is a vector whose components are all zero except for one of them. In order to create a one-hot vector for each word, first a dictionary[1] of all possible words in the available datasets is created. Assuming N words in the dictionary, an N-dimensional zero vector for each word is created and the component with the same index as the word in the dictionary is set to 1. Figure 2 demonstrates the one-hot vector for each word in a sample dictionary. Assuming the dictionary D has 5 words “I”, “cat”, “dog”, “have”, “a” sequentially with the indices 0–4, one-hot vector for each word is displayed in the figure.

The translation process is divided into the following two subprocesses.

2.3.1 Encoding Phase in MT

An RNN is used to extract a feature vector from the input sentence from the source language. All of the words in the input sentence are converted to one-hot vectors and passed to the RNN in the order of their presence in the sentence. The RNN then updates its hidden state and output vectors according to each word. The iteration is stopped when the End of Sentence (EOS) token is passed to the RNN. The EOS token is a token added manually to the end of input sentences to specify the end point of the sentence. The hidden state of the RNN after the EOS token is then used as the feature vector of the input sentence. One-hot vectors of the words in the input sentence are created using the dictionary of words from the source language.

[1] A dictionary is a list of unique words with unique indices.

2.3.2 Decoding Phase in MT

Another RNN is used to generate the words of the output sentence in an appropriate order. The decoder RNN is designed to predict a probability distribution over all possible words in the dictionary of the source language words at each step. Then a word is selected with respect to the produced probability distribution as the next word in the sentence. The iteration is stopped when the EOS token is generated by the decoder or a predefined number of words are generated.

The structure of the model proposed by Cho et al. [2] is shown in Fig. 3. The context vector extracted by the encoder is denoted by C, which is the hidden state of the RNN encoder at the last step.

2.4 Encoders in Machine Translation (Feature Extraction)

An RNN is used as the encoder in the model proposed by Cho et al. [2] Let h_e denote the hidden state of the encoder RNN. This state vector is updated at each time step t according to Eq. (1) in which h_e^t is the hidden state of the encoder at time step t, $f_{encoder}$ is a nonlinear activation function that can be as simple as an element-wise logistic sigmoid function and as complex as a Long Short-Term Memory (LSTM), and X_t is the one-hot vector of the tth word in the input sentence.

$$h_e^t = f_{encoder}\left(h_e^{t-1}, x_t\right) \tag{1}$$

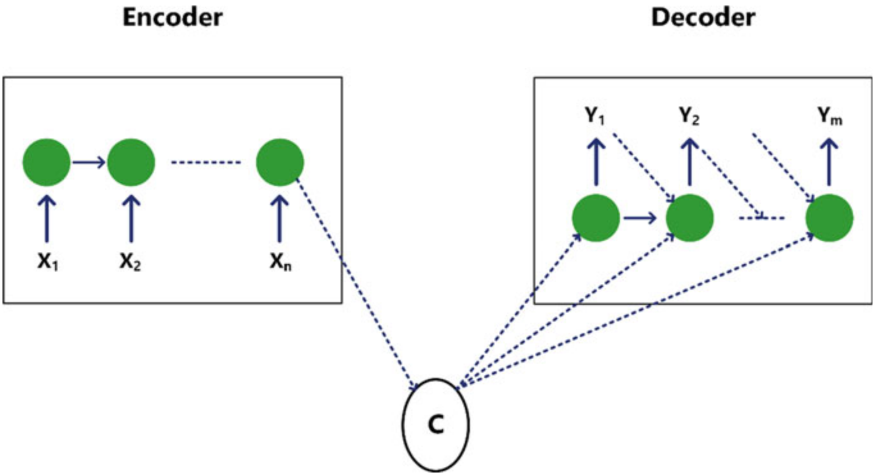


Fig. 3 An illustration of the first encoder-decoder based model proposed for machine translation

Assuming that the input sentence has Li words, the encoder RNN should iterate on each word and update its hidden state vector at each step. The hidden state of the RNN after the L_ith word is then passed to the decoder as the context vector C. So, the context vector extracted by the encoder can be computed as in Eq. (2).

$$C = h_e^{L_i} \tag{2}$$

2.5 Decoders in Machine Translation (Language Modeling)

The decoder is supposed to generate the output sentence word by word in a way that the meaning of the sentence is the same as the meaning of the input sentence represented by the context vector C. From another point of view, the decoder can be seen as an RNN that maximizes the likelihood of the translated sentence in the dataset for the input sentence and its generated context vector as expressed in (3), in which θ is the set of all trainable weights and the parameters of the model.

$$Pr_\theta\{Y|X\} \tag{3}$$

On the other hand, according to the encoder-decoder structure, the random variable C directly depends on the random variable X, and the random variable Y directly depends on the random variable C. Figure 4 displays the dependency graph between these 3 random variables.

Variable C is a "latent variable" since it is not directly observed, but is inferred by the model (specifically by the encoder). According to the dependencies displayed in Fig. 4 and considering the fact that the random variable Y directly depends on the latent variable C, the training procedure of the decoders can be separated from the training procedure of the encoders. In other words, since C is given while training the decoders, we can replace the likelihood expressed in (3) with the likelihood expressed in (4). Furthermore, assuming that each word in the sentence depends only on the meaning of the previous words in the sentence, the probability of a sentence can be replaced by the multiplication of the probabilities of its words given the previous ones.

$$Pr_\theta\{Y|C\} = \Pi_{t=0}^{Lo} Pr\{y_t|y_{t-1}, y_{t-2}, \ldots, y_0, C\} \tag{4}$$

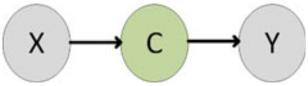


Fig. 4 Directed acyclic graph of dependencies of random variables in the encoder-decoder model

According to the Eq. (4) the training procedure of the encoder-decoder based models can be divided into two sub-procedures. First, the encoder is trained to extract the appropriate feature vector from the input. Then, the decoder is trained to generate the appropriate output given the feature vector extracted by the trained encoder. However, using the Eq. (4) allows the encoder and the decoder parts to be trained independently, they could be trained jointly in and end-to-end manner [2, 4, 5].

Another consequence of using the Eq. (4) is that the decoder is supposed to generate a probability distribution over each word at each step *t* given its previously generated words and the context vector extracted by the encoder. The probability distribution can be formulated by the RNN according to Eqs. (5) and (6). Let $\mathrm{h}_{\mathrm{d}}^{\mathrm{t}}$ be the hidden state of the decoder at time step *t*. Let O_{t} be the decoder output at time step *t* (an *Lo* dimensional vector) which is generated by a nonlinear function *g* applied on the decoder's hidden state and the context vector C. The decoder's hidden state is also generated by the nonlinear function $\mathrm{f}_{\mathrm{decoder}}$ applied on the previously generated word, hidden state of the decoder at previous time step and the context vector according to (7). With applying a *Softmax* on the output, a vector with the same size is generated whose sum of components is equal to one and can be treated as the desired probability distribution.

$$O^t = g\left(h_d^t, y_{t-1}, C\right) \tag{5}$$

$$Pr\{y_t|y_{t-1}, y_{t-2}, \ldots, y_0, C\} = SoftMax(O^t) \tag{6}$$

$$h_d^t = f_{decoder}\left(h_d^{t-1}, y_{t-1}, C\right) \tag{7}$$

At each time step, the probability distribution $Pr\left\{\mathrm{y}_{\mathrm{t}}|\mathrm{y}_{\mathrm{t-1}}, \mathrm{y}_{\mathrm{t-2}}, \ldots, \mathrm{y}_0, \mathrm{C}\right\}$ is generated by the decoder according to Eq. (6) and the next word is selected with respect to this probability distribution over the words in the dictionary of the destination language.

The two components of the proposed model can be jointly trained to minimize the negative conditional log likelihood expressed in (8) in which *N* is the number of samples in the dataset, Y_{n} and X_{n} are the *n*th output and input pair in the dataset, θ is the set of all trainable parameters, and *Loss* is the loss function to be minimized.

$$Loss = -\frac{1}{N}\Sigma_{n=0}^{N} \log Pr_\theta(Y_n|X_n) \tag{8}$$

3 Encoder Structure Varieties

The baseline encoder-decoder architecture proposed by Cho et al. [2] in machine translation attracted the attention of many researchers in different fields. As explained before, almost all of the variants of the baseline architecture in different tasks share a similar decoder, but the structure of encoder varies based on the type of input. In this section, we will introduce the important structures of encoders to encode different input types.

3.1 Sentence as Input

The simplest encoder for problems with sentences as inputs is an RNN. The first proposed encoder in machine translation is an LSTM which takes all words of the input sentence, processes them and returns the hidden state vector as the context vector.

Along with the RNNs, CNNs are employed to extract features from the source sentences in the encoding phase. As an instance, Gehring et al. proposed a convolutional encoder for machine translation in order to create better context vectors by taking nearby words into consideration using a CNN [6]. In this encoder, a CNN with a kernel size of k = 3 is used to extract a combination of each three nearby words' meaning in the sentence to generate the context vector.

In addition, different RNN cells are used as blocks of the encoder for sentence inputs. LSTMs [7] are widely used because of their ability to cope with long-term dependencies and remembering far history in the input sequence [5, 8–10]. GRU [2] is also used in different proposed models due to its good performance and the fact that it can be assumed as a light-weighted version of LSTM [2, 11–13].

The models proposed by Cho et al. [2] (RNNenc), Cho et al. [3] (grConv), Sutskever et al. [5] (Moses), Bahdanau et al. [8] (RNNsearch) are evaluated on an English to French translation task and the results are reported in Table 1. The BLEU score [14] is used to evaluate the machine translation models.

RNNenc model (proposed by Cho et al. [2]) uses the proposed RNN structure for encoding the input sentence while the grConv model (proposed by Cho et al. [3]) employs a gated recurrent convolutional network as the encoder and the Moses model (proposed by Krishevski [15]) uses LSTM cells as the encoder. Both of the first two models use gated recurrent units as the decoder while the third one uses the LSTM cells as the decoder.

According to the reported results, the Moses model outperforms the previous ones because of using LSTM cells which can cope with the problem of extracting long-term dependencies. In addition, results reported by Cho et al. [2] show that

Table 1 BLEU scores computed on the training and test sets

Model	BLEU score of training	BLEU score of testing
RNNenc	21.01	23.45
grConv	17.09	18.22
Moses	32.77	35.63
RNNsearch	–	36.15

the performance of the model highly decreases with the increments in length of the sentences. So, the main problem with the encoders and the decoders in machine translation tasks is extracting long-term dependencies. The model RNNsearch (proposed by Bahdanau et al. [8]) proposed a novel technique to cope with long-term dependencies called "attention mechanism" which will be introduced later in this paper. We will also discuss the challenge of "long-term dependencies" later in Sect. 4.1.

3.2 Image as Input

Encoder-Decoder based architectures form a majority of the proposed models to generate captions for images. In such models, the process of generating captions for the input image is divided into two steps. The first step is encoding in which a feature vector extracted from the image is returned as the context vector. The second step is decoding in which the generated context vector is passed to a decoder to generate sentences describing the context.

The best choice for encoders in such problems is a CNN. Almost all of the proposed models for image captioning based on the encoder-decoder framework use different types of CNNs as the encoders. Neural encoder-decoder based approaches to image captioning share the same structure for decoder, while in most of them the encoder consists of a single CNN. So, the extracted feature vector from the image can be expressed in Eq. (9) in which X is the input image, CNN(X) is the output of the CNN network, and C is the context vector passed to the decoder.

$$C = CNN(X) \tag{9}$$

A wide variety of CNNs are employed as encoders in the proposed models for image captioning. Since the pretrained versions of VGGNet [16] and AlexNet [15] on the ImageNet dataset [17] extract good features from images for different tasks and are available online, they have been used as encoders in different proposed image captioning models [18–20]. Furthermore, ResNet [21] has been widely used because of its good performance as the encoder in such models [22–24]. Google NIC Inception v3 [25] has also been used in proposed models because of its better image classification accuracy compared to ResNet [26–29]. Yao et al. [30] integrated attribute based LSTMs (LSTM-A) with the CNNs and trained them in an end-to-end manner to boost the encoder.

Figure 5 illustrates the use of CNNs as the encoder in models based on the encoder-decoder framework for image captioning proposed by Vinyals et al. [28] Similar architectures are used to generate captions in other studies. As it is shown in the figure, the encoder part of the model consists of a CNN extracting a feature vector from the input image. The extracted feature vector is then passed to the decoder to generate the appropriate caption. The decoder consists of an RNN which generates the probability of the next word according to (4) at each step.

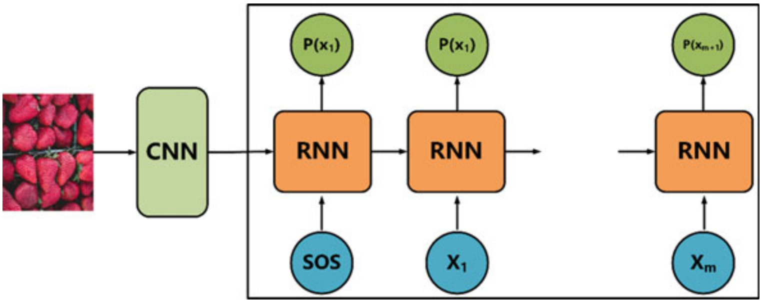


Fig. 5 Model architecture based on encoder-decoder framework for image caption generation

Table 2 Performance of the encoder-decoder based models on MSCOCO dataset

Model	B-1	B-2	B-3	B-4	METEOR	CIDEr	ROUGE$_L$
DeepAlign	62.5	45.0	32.1	23.0	19.5	66.0	–
SCA-CNN	71.9	54.8	41.1	31.1	25.0	–	–
PG	75.4	59.1	44.5	33.2	25.7	101.3	55.0
NIC	–	–	–	27.7	23.7	85.5	–
VDD	73.7	66.4	57.1	50.5	34.7	125.0	64.9

The models proposed by Karpathy et al. [18] (DeepAlign), Chen et al. [19] (SCA-CNN), Vinyals et al. [28] (NIC), Liu et al. [29] (PG), and Asadi et al. [31] (VDD) are evaluated on a popular image captioning dataset proposed by Lin et al. [32] called MSCOCO. The proposed models are evaluated using the BLEU scores, METEOR score [33], CIDEr score [34], and ROUGE$_L$ score [35] and the results are reported in Table 2. All of these models used CNNs as the encoder.

3.3 Video as Input

Another sort of problems that the encoder-decoder models play an important role in solving them are those with videos as the input and describing text as the output, also called "Video Description Generation" or "Video Captioning". Since creating a good representation is critical to the overall performance of video captioning models, a wide variety of encoders are proposed to cope with different difficulties and challenges of such systems. This section presents some examples of encoders proposed to deal with the challenges of extracting motion details from the video.

Assume an input video V consists of Li frames. We can present the video as in Eq. (10), in which $\mathrm{v_I}$ is a representation of the ith frame in the input video and $\mathrm{v_{Li}}$ is the end of video token (<EOV>). In fact, each $\mathrm{v_I}$ is the feature vector extracted by a CNN on the ith frame in the input video.

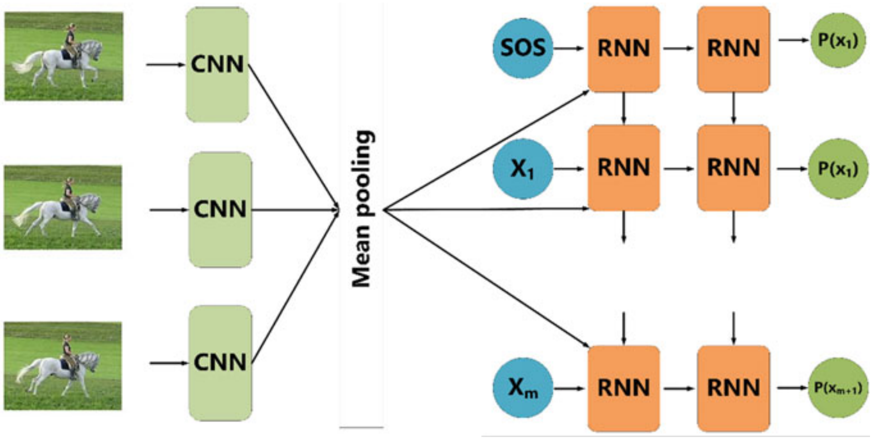


Fig. 6 An illustration of the first encoder-decoder based model for video captioning

$$V = \{v_0, v_1, \ldots, v_{Li}\} \tag{10}$$

Since in the baseline encoder-decode model, the encoder should return a “fixed length” context vector extracted from the input, an aggregation function is required to aggregate feature vectors from different frames in the video and pass it as the context vector to the decoder.

Different ideas have been employed to propose a good aggregation for video captioning. The first end-to-end encoder-decoder based approach in video description generation proposed by Venugopalan et al. in 2014 [4] used a mean pooling layer to create the fixed length context vector from the input video. In that model, first a CNN is applied to each frame of the input video. Then a mean pooling layer is applied to create an average feature vector over the set of feature vectors extracted from each frame. The average feature vector is then passed to the decoder to generate the sentence. A stacked RNN structure is used as the decoder. Figure 6 demonstrates the architecture of the first encoder–decoder based model for video captioning [4].

Different CNNs have been used to extract feature vectors from the frames of the input video. For instance, Majd et al. [36] proposed an extended version of the LSTM cells, called “C2LSTM” in which the motion data as well as the spatial features and the temporal dependencies are perceived by embedding a correlation modeling layer into the cell. Majd et al. [37] also proposed a novel network architecture using previously proposed C2LSTM as the encoder for human action recognition.

3.3.1 3D-CNNs

Extracting good features from the input video is a challenging task that can highly affect the performance of the proposed model. The extracted context vector from the input video should well express the detailed motions in the video. In order to create

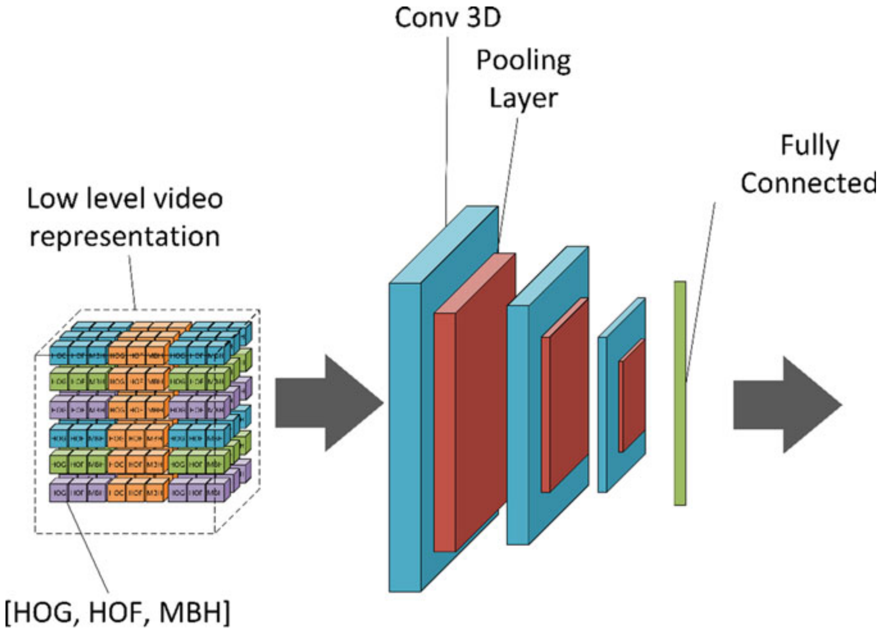


Fig. 7 The structure of 3D-CNN

an encoder capable of extracting fine motion features from the video, Yao et al. [38] proposed a 3D-CNN as the encoder. The structure of this 3D-CNN is illustrated in Fig. 7.

Actually, the proposed 3D-CNN models the spatio-temporal dependencies in the input video. The 3D-CNN is used to build a higher-level representation that preserves the local motion information from short frame sequences in the input video. This is accomplished by first dividing the input video clip into a 3D spatio-temporal grid of 16 * 12 * 2 (width * height * timesteps) cuboids. Each cuboid is represented by concatenating the histogram of oriented gradients (HOG), histogram of oriented flow (HOF) and motion boundary histogram (MBH) with 33 bins. This transformation ensures that the local temporal structures and motion features are well extracted. The generated 3D descriptor then is passed to 3 convolutional layers each followed by a max-pooling layer and one fully connected layer followed by a softmax layer as demonstrated in the Fig. 7. The output of the 3D-CNN is then passed to the decoder to generate an appropriate caption.

The 3D-CNN proposed by Yao et al. [38] is also used along-side the typical 2D-CNN in other works. Pan et al. [39] proposed a novel encoder-decoder architecture for video description generation and used the 3D-CNN and the typical 2D-CNN and applied a mean pooling layer to the set of features extracted by each of the CNNs and concatenated the output to generate the context vector of the video. Figure 8 illustrates the encoder part of this model.

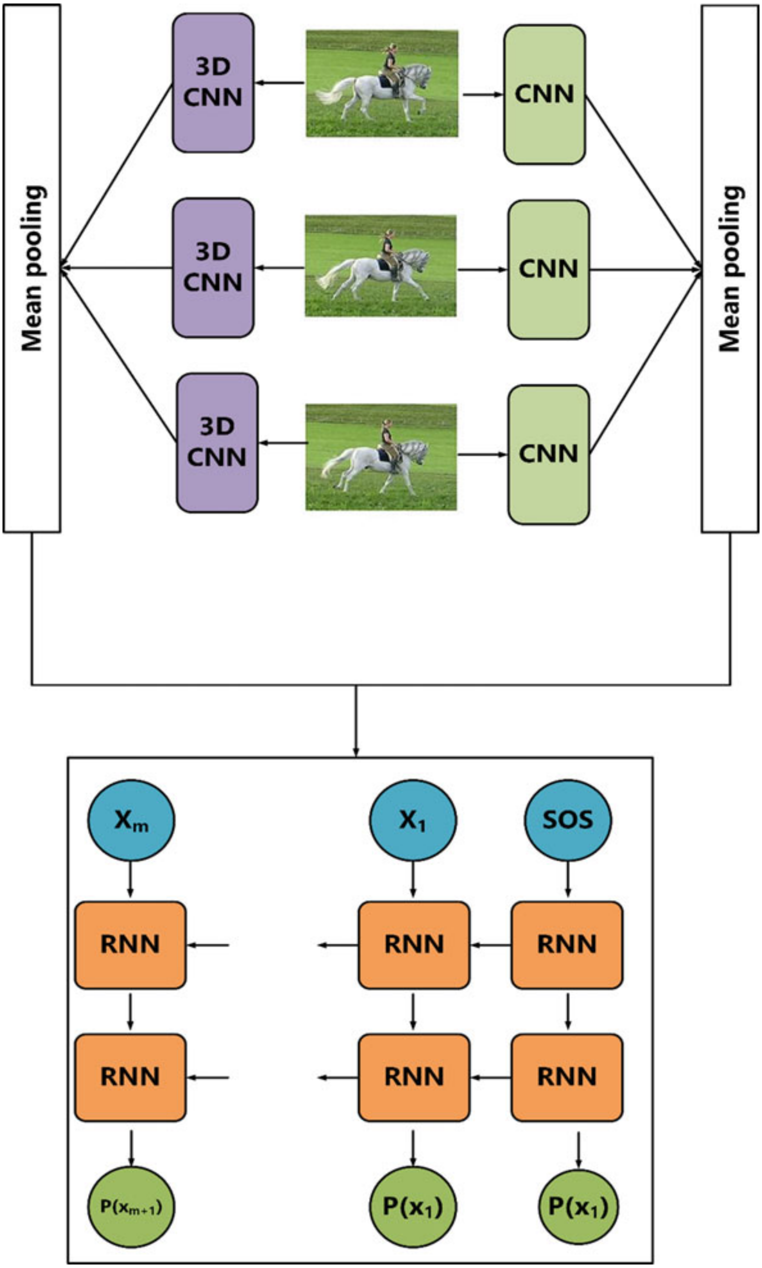


Fig. 8 An illustration of the encoder structure which uses a combination of 2D CNNs and 3D CNNs

3.3.2 Dense Video Captioning

Another approach to video captioning includes those methods focusing on “Dense Video Captioning”. Despite the models that generate a single sentence as the description of the input video, dense video captioning models first detect and localize the existing events in the input video and then generate a description sentence for each of the detected events.

Encoders for dense video captioning are supposed to first detect all of the existing events in the input video. Then for each of the events a quadruple $< t_{start}, t_{end}, score, h >$ should be extracted. t_{start} and t_{end} are the starting and ending frame numbers of the specified event. score is the confidence score of the encoder for each of the events. If the score of an event is greater than a threshold, it is reported as an event and its quadruple is passed to the decoder for sentence generation; otherwise, it is ignored. Finally, h is the feature vector extracted from the range of frames between t_{start} and t_{end} which is used by the decoder as the context vector of the event to generate a sentence for the event [40].

The task of dense video captioning was proposed by Krishna et al. [41] first in 2017. The proposed encoder by Krishna et al. [41] for dense video captioning is able to identify events of the input video within a single pass while the proposed decoder simultaneously generates captions for each event detected and passed by the encoder.

Figure 9 illustrates the structure of encoder proposed by Krishna et al. [41] for dense video captioning. The proposed encoder is able to extract all events in the input video using a deep action proposal (DAP) module proposed by [42]. To do this, a 3D-CNN is applied to the input video frames to extract video features. These video features are passed to the DAP module. This module consists of different LSTMs that are applied to the video features sequence in different resolutions and are trained to detect starting and ending points of events. The confidence score of each event is

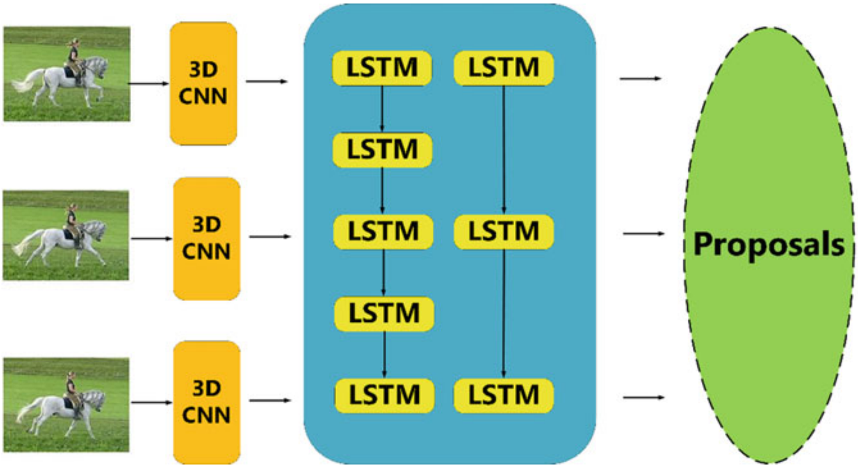


Fig. 9 An illustration of the encoder model for dense video captioning

also computed by DAP. The proposed event proposals are then sorted with respect to their ending points and passed sequentially to the decoder. The feature vector of each event is also the hidden state of the corresponding RNN in the DAP. The decoder then generates a sentence for each event using its feature vector as the encoder output.

Li et al. [40] proposed a novel end-to-end encoder-decoder based approach for dense video captioning which unified the temporal localization of event proposals and sentence generation. Figure 10 illustrates the structure of the proposed model [40]. Here, instead of using an extra DAP module, a 12-layer convolutional structure is designed to extract features for action proposal over the output of the 3D-CNN. The first 3 layers of the convolutional structure (500D layer and base layers in Fig. 10) are designed to introduce nonlinearities and decrease the input dimension. The next 9 layers, which are called “Anchor layers”, extract features from different resolutions to be used for event prediction. The “Prediction Layer” consists of three parallel fully connected layers to first regress temporal coordinates ($\mathrm{t}_{\mathrm{start}}$ and $\mathrm{t}_{\mathrm{end}}$) of each event, then compute the descriptiveness of the event (score) and finally classify the event vs background. The prediction layer is applied to the output of all anchor layers to enable the model to detect events from different resolutions. The extracted proposals are then passed to the proposal ranking module which ranks event proposals with respect to their ending time. Finally, the events are passed to the decoder for sentence generation sequentially.

A wide variety of models are proposed to cope with the difficulties of the encoding phase in dense video captioning. Shen et al. [43] proposed a new CNN called “Lexical FCN” which is trained in a weakly supervised manner to detect events based on the captions in the dataset. Duan et al. also proposed a novel approach for dense video captioning based on the similar assumption “each caption describes one temporal segment, and each temporal segment has one caption” [44]. Xu et al. proposed an end-to-end encoder-decoder based model for dense video captioning which detects and describes events in the input video jointly and is applicable to dense video captioning on video streams. Zhou et al. also proposed an end-to-end approach with a masking network to localize and describe events jointly [45]. Wang et al. proposed a novel architecture to take both past and future frames into account while localizing the events in the input video using [46] bidirectional models.

The models proposed by Venugopalan et al. [4] (LSTM-YT), Venugopalan et al. [47] (S2VT), Yao et al. [38] (3D-CNN), and Pan et al. [39] (LSTM-E) for video captioning are evaluated on the Youtube2Text dataset proposed by Chen et al. [48], the results of which are reported in Table 3.

The LSTM-YT and S2VT models use similar encoders. In both of these models, a CNN is used to extract a feature vector from each frame in the video. The extracted feature vectors are then passed to a mean-pooling layer in order to generate a unified feature vector to represent the input video. In 3D-CNN model, a 3D-CNN is used along with a 2D-CNN to extract and represent information about the movements in the input video. The extracted feature vector in this model contains spatio-temporal information extracted from the video. The LSTM-E model used LSTM cells as the encoder. As a result, the extracted feature vector represents the temporal information of the input video.

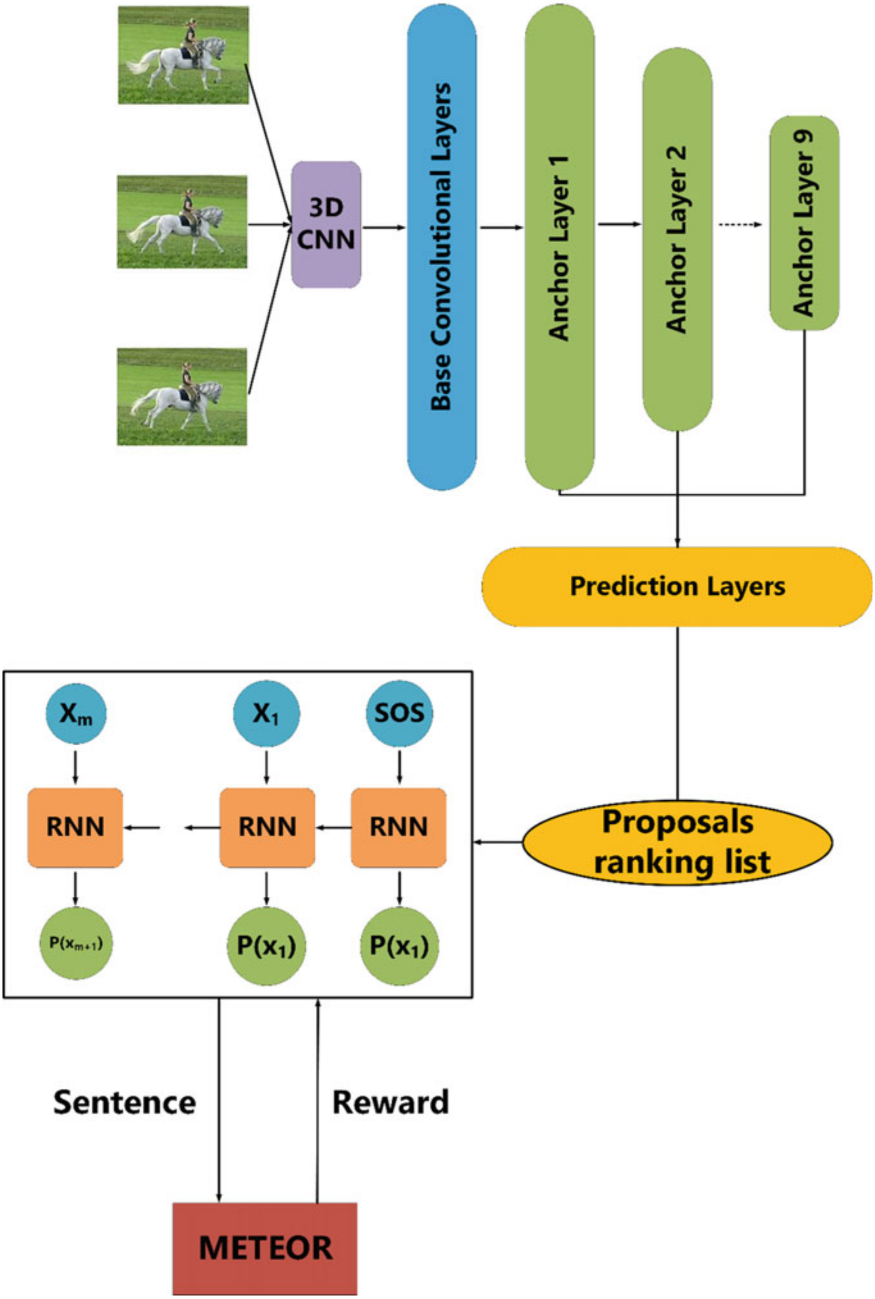


Fig. 10 An illustration of the encoder-decoder structure for dense video captioning

Table 3 Performance of different encoder-decoder based models for video captioning

Model	B-1	B-2	B-3	B-4	METEOR	CIDEr
LSTM-YT	–	–	–	33.29	29.7	–
S2VT	–	–	–	–	29.8	–
3D-CNN	–	–	–	41.92	29.6	51.67
LSTM-E	78.8	66.0	55.4	45.3	31.0	–

The main challenging problem in the encoders of the encoder-decoder based models in video description generation, is to extract a combination of the spatial and the temporal information of the input video.

4 Decoder Structure Varieties

In the encoder-decoder based models, decoders generate a sequential output for the given input. The generated output might be in the form of a descriptive text (the desired output in machine translation, image/video captioning, textual/visual question answering, and speech to text conversion), or a speech signal (the desired output in the text to speech challenge). The output is a numerical sequence that is passed to the last layer in order to generate an appropriate output for the given input. Therefore, the main structures of the decoders are similar in different tasks. This section, introduces different techniques proposed to make better decoders with better generated captions.

4.1 Long-Term Dependencies

One of the basic problems with RNNs is the problem of "long-term dependencies". Indeed, when the length of the input or the length of the desired output is too large, the gradients in these networks should propagate over many stages. When the gradient is propagated over a large number of stages, it tends to either vanish or explode.

In addition, the gradients in each backpropagation step are multiplied by small coefficients or small learning rates. Thus, the gradient in the early stages will be close to zero and might make no significant change in the weights of the early stage layers [49].

In this section we will discuss the approaches proposed to cope with the long-term dependency challenge in the decoders.

4.2 LSTMs

LSTMs have achieved excellent results on a variety of sequence modeling tasks thanks to their superior ability to preserve sequence information over time. The combination of the "memory cell" and the "forget gate" in the structure of LSTM improves its ability to model sequence information by training to forget the unnecessary information (using the forget gate) and keep the necessary information in the memory cell. Cho et al. [2], Bahdanau et al. [8], Luong et al. [50], Wu et al. [51], Johnson et al. [52] and Luong et al. [53] used LSTMs as both the encoder and decoder part of their models proposed for machine translation.

4.3 Stacked RNNs

As mentioned earlier, multi-staged decoders are hard to train due to the vanishing gradient problem. Thus, most of the proposed encoder-decoder based models use a single layer RNN as the decoder which results in difficulties to generate rich fine-grained sentences. Stacking multiple RNNs on top of each other is another way to enable decoders to generate sentences describing more details of the input image.

Donahue et al. [54] proposed an encoder-decoder based approach to image captioning which uses a stacked structure of LSTMs as the decoder in order to describe more details of the input image. In this method an LSTM is used on top of another one in a way that the first layer LSTM takes image features and the previously generated word embedding along with its previous hidden state vector as the input and generates a coarse low-level representation of the output sentence. In the next step, the hidden state of the low-level LSTM is passed to the next LSTM as the input along with its previous hidden state to generate the fine high-level representation of the output. A *softmax* layer is then applied to the generated high-level representation of the output to generate the probability distribution of the next word in the sentence. Gu et al. [55] also used encoder-decoder based model with a two-layer stacked LSTM as the decoder in order to enable the proposed model to generate better descriptions.

The idea of employing a stacked structure of RNNs as the decoder is also used in models proposed for video description generation. Venugopalan et al. [4] proposed the first decoder in neural encoder-decoder based approaches for video description generation with a simple stacked structure. Figure 11 demonstrates the architecture of a sample stacked decoder. Blocks tagged with "C" display the input at each step. The red line illustrates the shortest path from the first step to the output in the model. Since the length of the shortest path from the first step to the output correlates with the testing and the training time of the model, decreasing this length decreases the testing and the training time of the model.

Along with the methods using stacked RNNs as decoders, a category of models is proposed which follow a hierarchical fashion to arrange RNNs in decoders in order to enable the models to generate fine-grained output sequences.

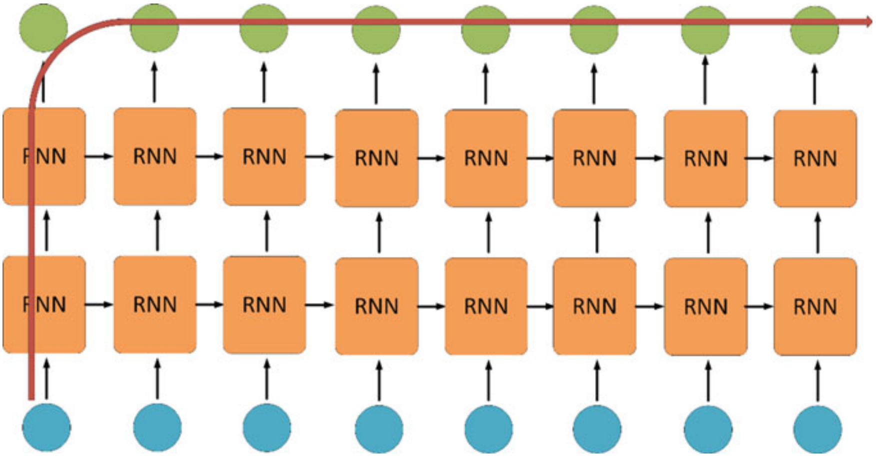


Fig. 11 Stacked structure of RNNs

In addition, hierarchical RNN structures are also used to enable encoders in problems with a sequential input to exploit and encode more detailed information from the input. Pan et al. [56] proposed an encoder-decoder based model for video description generation with a hierarchical encoder structure. In their model, two layers of different LSTMs are used. The first layer LSTM is applied to all sequence steps in order to exploit low-level features and the second layer LSTM is applied on the output of equally sized subsets of the input sequence steps to exploit the high-level features. Figure 12 demonstrates this architecture. The illustrated red line, shows the shortest path from the first step to the output. Comparing structures displayed in Figs. 11 and 12 shows that the shortest path from the first step to the output in hierarchical

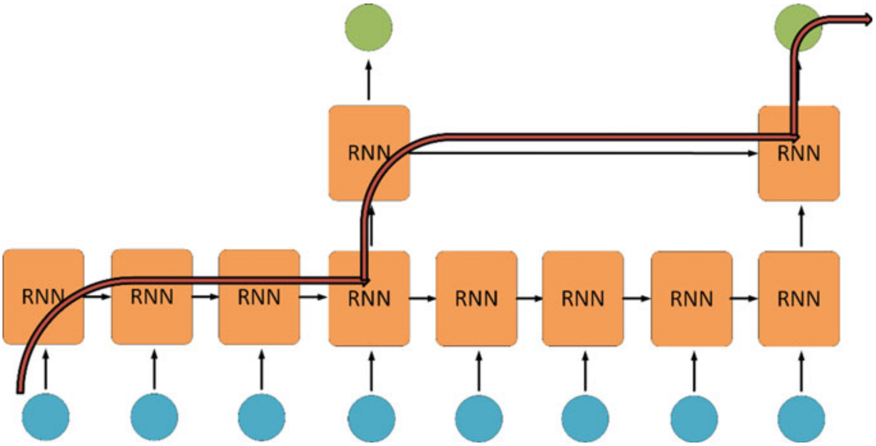


Fig. 12 Hierarchical structure of RNNs

Table 4 Performance of different encoder-decoder based models with stacked decoder structure in video captioning

Model	B-1	B-2	B-3	B-4	METEOR	CIDEr
LSTM-YT	–	–	–	33.29	29.7	–
HRNE	79.2	66.3	55.1	43.8	33.1	–
h-RNN	81.5	70.4	60.4	49.9	32.6	65.8

models is much smaller than that in stacked models. Therefore, the efficiency of the hierarchical model is much higher than that of the stacked model.

More complex hierarchical structures are also proposed in the literature for different intents. Yu et al. [57] proposed a model with a hierarchical structure to generate a set of sentences arranged in a single paragraph as a description for the input video. The first layer in this model is a simple decoder to generate single sentences and the second layer is a “paragraph controller”. The paragraph controller is another RNN which generates a feature vector given the last hidden state of the first layer RNN denoting the meaning of the next sentence to be generated. The first layer RNN then takes the feature vector generated by the second layer and concatenates it with other inputs to control the meaning of the next sentence.

The models proposed by Venugopalan et al. [4] (LSTM-YT), Pan et al. [56] (HRNE), and Yu et al. [57] (h-RNN) are evaluated on Youtube2Text dataset proposed by Chen et al. [48]. Table 4, reports the evaluation results on this dataset.

The LSTM-YT model, uses a simple 2-layer stacked decoder to generate appropriate caption for the input video. The HRNE model, uses a hierarchical decoder structure to reduce the length of the shortest path from the input to the output of the decoder. The h-RNE model uses two-steps, one of which generates a sentence and the other one controls the paragraph context.

According to the results reported in Table 4, the METEOR score of the HRNE and the h-RNN models are similar, while they are better than that of the LSTM-YT model. The results indicate that hierarchical decoder structures are better in extracting the long-term dependencies from the input than the stacked decoders.

4.4 *Vanishing Gradients in Stacked Decoders*

Even though increasing the depth of the stacked decoder structure adds more nonlinearities to the model and empowers it to generate fine-grained sentences, the number of layers in the stacked structures is strictly restricted. Most of stacked decoder structures use at most 2-layers of RNNs on top of each other [54, 56, 57].

Indeed, the most important issue restricting the number of layers in stacked structures is the problem of vanishing gradients in deeper decoders. The backpropagated gradients vanish as a result of two facts. First, the gradients in such architectures are multiplied by small multipliers and small learning rates at each stage. Second, since the loss function of the proposed decoders is based on the likelihood of the next word, decoders are supposed to predict a probability distribution over all the words in the dictionary. It means the decoder's output size is equal to the size of word dictionary. Furthermore, the sum of all components in the output layer is supposed to be equal to 1, which means the gradients computed at the last layer are numerically small. Summing up, the gradients vanish in stacked decoders since the computed gradients at the last layer are small and they are multiplied by small multipliers at each step.

Asadi et al. [31] proposed a novel approach to train the stacked decoders in a way that the gradients of the last layer are large enough to make significant changes in the weights of the first layers. The main idea is to use a word-embedding vector instead of a one-hot vector representation as the decoder desired output. As a result, the optimization problem changes from predicting the conditional probability distribution of the next word to a word-embedding regression. In this way, the limitations of the value of the computed gradients at the last step are resolved. In addition, the loss function of the decoder is changed from the cross-entropy to MSE of the word embedding of the next word.

To shed light over the issue, we recall the cost function $Loss(Y_i, D_i)$ of the baseline encoder-decoder model proposed for machine translation by Cho et al. [2]. Let Y_i be the probability distribution predicted by the decoder for the input sentence X_i, and let D_i be the desired output for the given input. The proposed loss function is based on the *cross-entropy* loss function which is typically used for classification. The optimization problem (11) can be used to train the model. This optimization problem, determines the trainable parameters of the model θ in a way that while the model generates a probability distribution Y_i for the input X_i, the classification error of the model over the dataset is minimized. Note that in this problem, λ is a regularization parameter controlling the size of the trainable parameters of the model and N_x is the number of records in the dataset.

$$\begin{aligned} minimize \sum\nolimits_{i=0}^{N_x} Loss(Y_i, D_i) + \lambda|\theta|^2 \\ subject\, to : |Y_i|^2 = 1 \end{aligned} \tag{11}$$

One way to create higher gradients at the last layer of the decoder is to remove the *Softmax* layer from the top of the decoder. If this layer is removed, the generated output of the model Y_i no more is in the form of a probability distribution. Therefore, the optimization model could not be formulated as shown in (11).

Asadi et al. [31] proposed a model to cope with this problem by changing the optimization problem (11). In the proposed model, the task of generating sentences in a word by word manner is treated as a regression rather than a classification task. Asadi et al. [31] augmented the model with an embedding function E which generates an embedding vector for each word in the dictionary and returns the most similar word in the dictionary given an embedding vector. Using this embedding function, the optimization problem (11) could be changed from predicting the probability distribution of the next words to regressing the embedding vector of the next word in the sentence. So, the optimization problem is changed to (12).

Table 5 Performance of different techniques to cope with the problem of vanishing gradients in encoder-decoder based models for image captioning

Model	B-1	B-2	B-3	B-4	METEOR	CIDEr	ROUGE$_\mathrm{L}$
VDD	73.7	66.4	57.1	50.5	125.0	34.7	64.9
StackedCap	78.6	62.5	47.9	36.1	120.4	27.4	–
SOT	74.3	57.9	44.3	33.8	104.4	33.8	54.9
WeightedTrain	76.8	60.5	45.8	34.2	105.5	26.1	55.5

$$minimize \frac{1}{N_x} \Sigma_{i=0}^{N_x} (E(Y_i) - E(D_i))^2 + \lambda|\theta|^2 \tag{12}$$

As Eq. (12) shows, the constraint on the size of the output is omitted because the model output is no longer in the form of a probability distribution. In addition, as the problem is changed from a classification to a regression task, the *cross-entropy* loss is replaced with the *mean squared error*. The model proposed by Asadi et al. [31] is applied on the decoder of an encoder-decoder model for image captioning, and it outperforms the state-of-the-art models in the field.

The models proposed by Asadi et al. [31] (VDD), Gu et al. [55] (StackedCap), Chen et al. [58] (SOT), and Ding et al. [59] (WeightedTrain) are evaluated on the MSCOCO dataset and the results are reported in Table 5.

The StackedCap model used a stacked structure that generates captions for the image in a coarse to fine manner. In this model, the decoder consists of multiple LSTMs, each of which operating on the output of the previous one. This decoder generates increasingly refined captions for the input image. The SOT model used an attribute-based attention mechanism to cope with the long-term dependencies. Finally, the WeightedTrain model added some reference knowledge to help the decoder to generate more descriptive captions. In this model, each word is assigned a weight according to the correlation between that word and the input image. In this way, the decoder can attend more to the important words while generating captions.

According to the results reported in Table 5, treating the sentence generation as a regression problem rather than a classification one empowers the decoders to generate better longer sentences. The VDD model outperforms other state-of-the-art models with respect to the CIDEr, METEOR, ROUGE$_\mathrm{L}$, and BLEU scores except for BLEU-1.

4.5 Reinforcement Learning

One of the problems of training the decoders using the loglikelihood model is that the performance of the model is highly different on the training and testing sets. This occurs since the optimization function for training is different from the evaluation metrics used in testing. Recently, reinforcement learning has been used to decrease the gap between training and testing performance of the proposed models. In other words, the main problem with the loglikelihood objective is that it does not reflect the task reward function as measured by the BLEU score in translation.

Wu et al. [51] proposed the first decoder trained by reinforcement learning for machine translation. After that, other researchers used reinforcement learning to train decoders in other tasks. Wang et al. [60] proposed the first decoder trained with reinforcement learning, taking CIDEr [34] score as the reward in video captioning. Li et al. [40] also trained a decoder in a reinforcement learning fashion using the METEOR [33] score as the reward to generate caption for the input videos.

Figure 13 illustrates the structure of the model proposed by Wang et al. [60] The decoder in this work consists of three different modules, namely a manager, a worker, and an internal critic. These three modules are trained using a reinforcement learning method. The manager operates at a lower temporal resolution and emits a goal when needed for the worker, and the worker generates a word for each time step by following the goal proposed by the manager. The internal critic determines if the worker has accomplished the goal and sends a binary segment signal to the manager to help it update goals.

Figure 14 illustrates the unrolled decoder proposed by Wang et al. [60] The manager takes the context vector c_t^M at time step t and the feature vector of sentence generated at previous time step h_{t-1}^W as the input. An LSTM is used to model the

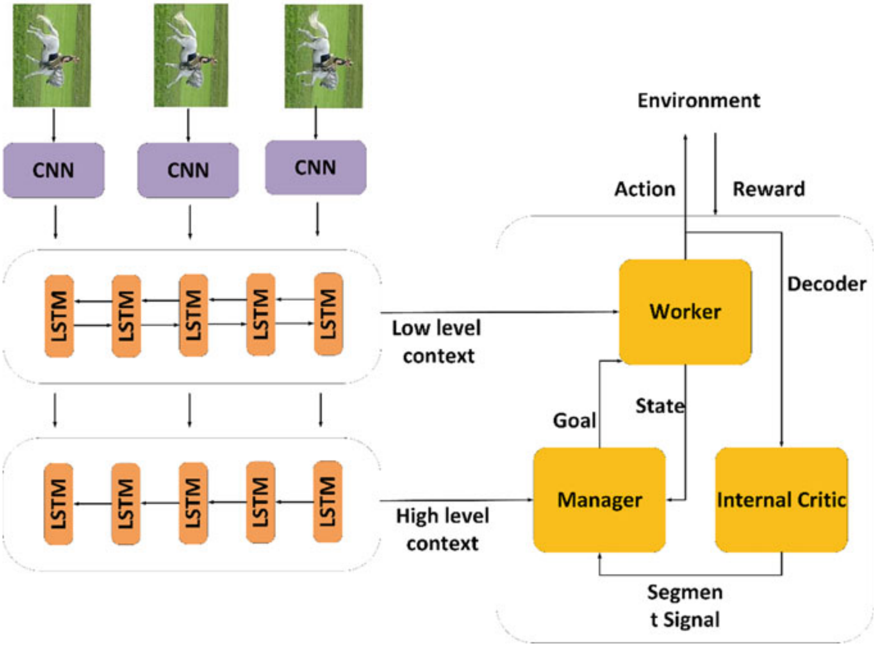


Fig. 13 An illustration of the encoder-decoder based model in which the decoder is trained using reinforcement learning

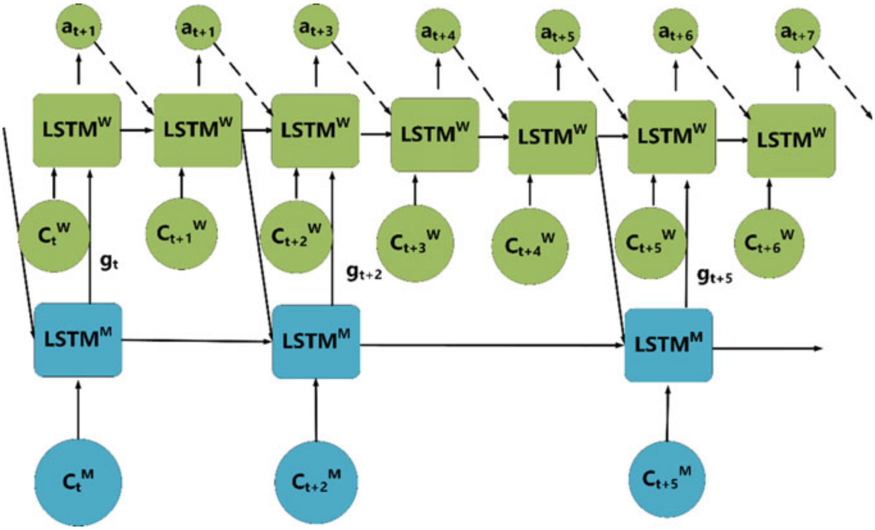


Fig. 14 An illustration of the unrolled decoder

extracted goal sequences. The LSTM takes the input and updates its hidden state $\mathrm{h}_{\mathrm{t}}^{\mathrm{M}}$. The hidden state of the LSTM is then used to generate the next goal using the nonlinear function u_{M} according to (13).

$$h_t^M = LSTM^M(h_{t-1}^M, [c_t^M, h_{t-1}^W]) \tag{13}$$

$$g_t = u_M(h_t^M) \tag{14}$$

$LSTM^M$ denotes the LSTM function used in manager, u_M is the function projecting hidden states to the semantic goal, $\mathrm{h}_{\mathrm{t-1}}^{\mathrm{M}}$ is the hidden state of the manager LSTM at the previous time step, and g_{t} is the vector of semantic goal generated at time step t.

The worker then receives the generated goal g_t, takes the concatenation of $[c_t^W, g_t, \alpha_{t-1}]$ as the input, and outputs the probabilities π_t over all actions $\alpha t \in V$, where each action is a generated word according to Eqs. (15)–(17).

$$h_t^W = LSTM^W(h_{t-1}^W, [c_t^W, g_t, \alpha_{t-1}]) \tag{15}$$

$$x_t = u_W(h_t^W) \tag{16}$$

$$\pi_t = SoftMax(x_t) \tag{17}$$

The internal critic is used to provide a good coordination between the manager and the worker. Internal critic is indeed a classifier to determine when the worker is done with generating an appropriate phrase for a given goal. When the worker is done, the internal critic sends an activation signal to the manager to generate a new goal. Let $\mathrm{z_t}$ be the binary signal of the internal critic, the probability $\mathrm{Pr(z_t)}$ is computed according to Eqs. (18) and (19).

$$h_t^I = LSTM^I([h_{t-1}^I, \alpha_t]) \tag{18}$$

$$\Pr(z_t) = sigmoid\big(W_z h_t^I + b_z\big) \tag{19}$$

The objective of the worker is to maximize the discounted return in which θ_W is the set of trainable parameters of the worker, γ is the discount rate, and r_{t+k} is the reward at step $\mathrm{t + k}$. Therefore, the loss function of the decoder can be written as (21).

$$R_t = \Sigma_{k=0}^{\infty}\gamma^k r_{t+k} \tag{20}$$

$$L(\theta_W) = -E_{\alpha_t \sim \pi_{\theta_W}}[R(\alpha_t)] \tag{21}$$

The gradient of the non-differentiable, reward-based loss function can be derived as:

$$\nabla_{\theta_W} L(\theta_W) = -E_{\alpha_t \sim \pi_{\theta_W}}[R(\alpha_t \nabla_{\theta_W} \log \pi_{\theta_W}(\alpha_t))] \tag{22}$$

Typically, the expectation of the loss function if estimated with a single sample, so the expectation term can be omitted. In addition, the reward can be subtracted with a baseline $\mathrm{b_tW}$ in order to generalize the policy gradient.

$$\nabla_{\theta_W} L(\theta_W) \approx -(R\big(\alpha_t - b_t^W\big)\nabla_{\theta_W} \log \pi_{\theta_W}(\alpha_t) \tag{23}$$

The manager is supposed to be trained in a way that it can compute goals to generate sentences with better BLEU scores. The action of the decoder is produced by the worker. So, the worker is assumed to be fully trained and used as a black box when training the manager. More specifically, the manager outputs a goal $\mathrm{g_t}$ at step t and the worker then runs c steps to generate the expected segment $e_{t,c} = \mathrm{\alpha_t\alpha_{t+1}\alpha_{t+2}} \cdots \mathrm{\alpha_{t+c}}$ using the goal. Then the environment responds with a new state $\mathrm{s_{t+c}}$ and reward $\mathrm{r(e_{t,c})}$. Following a similar math, the final gradients for training the manager can be derived as in (24).

$$\nabla_{\theta_M} L(\theta_M) = -\big(R\big(e_{t,c}\big) - b_t^M\big)\big[\Sigma_{i=t}^{t+c-1}\nabla g_t \log \pi(\alpha_i)\big]\nabla_{\theta_M}\mu_{\theta_M}(s_t) \tag{24}$$

In which $\mathrm{\mu_{\theta_M}(s_t)}$ is a noisy version of the generated goal and is used in order to empower exploration in the training of the model. Furthermore, rewards are defined as (25) and (26).

$$R(a_t) = \Sigma_{k=0}^{\infty}\gamma^k\big[CIDEr(sent + \alpha_{t+k}) - CIDEr(sent)\big] \tag{25}$$

$$R(e_t) = \Sigma_{n=0}^{\infty}\gamma^n[CIDEr(sent + e_{t+n}) - CIDEr(sent)] \tag{26}$$

Other metrics such as BLEU score can be used instead of CIDEr.

5 Attention Mechanism

Models based on the encoder-decoder framework encode input to a "fixed length vector". The decoder in these models generates the output based on the information represented in the fixed length encoder output. Each element of the output may be more strongly related to a specific part of the input. In these cases, more detailed information about that specific part of the input is required, and the extra information from the other parts of the input could deceive the model.

Attention mechanism, first introduced by Bahdanau et al. [8] in machine translation, is a mechanism that allows the encoder-decoder models to pay more attention to a specific part of the input, while generating the output at each step. Furthermore, the mechanism enables decoders to cope with the long-term dependencies and generate more fine-detailed sentences and outputs.

In this section, first the basic idea of the attention mechanism proposed in machine translation is described. Then, the use of this mechanism in some encoder-decoder architectures proposed in various applications is discussed.

5.1 Basic Mechanism

Bahdanau et al. [8] proposed the first encoder-decoder based model equipped with the attention mechanism in order to produce better translations. The encoder and the decoder parts of the proposed model are changed. The encoder is modified to generate a sequence of feature vectors called "annotation vectors" and an extra layer called "attention layer" is added in between the encoder and the decoder. The attention layer receives the annotation vectors generated by the encoder and creates a fixed length context vector at each step and passes it to the decoder in order to generate the probability of the next word in the sentence.

Based on these changes, the target probability distribution of the decoder can be expressed as in (27). It denotes the probability of the next word $\mathrm{y_t}$ at time step t, given all of the previously generated words and the context vector generated to predict the tth word. The decoder computes this probability at each step.

$$Pr(y_t|y_{t-1}, \ldots, y_0, C_t) \tag{27}$$

Let $L = \{l_0, l_1, \ldots, l_{N_i}\}$ be the set of generated annotation vectors by the encoder, and N_i be the number of generated annotations, the context vector C_t is then computed at each step using the Eq. (28). The coefficients α_k in (28) are called the "attention weights".

$$C_t = \Sigma_{k=0}^{N_i} \alpha_k^t l_k \tag{28}$$

The key point in generating the context vector at each step using the attention mechanism is to compute the attention weights at each step. Researchers have proposed different ways to compute the attention weights. One of the most used attention mechanisms, which is called "Soft Attention", is proposed by Xu et al. [10]. The attention weights in soft attention are computed using Eqs. (29) and (30).

$$\alpha_k^t = \frac{\exp\left(e_k^t\right)}{\Sigma_{j=0}^{N_i} \exp\left(e_j^t\right)} \tag{29}$$

$$e_k^t = f(h_{t-1}, l_j) \tag{30}$$

Equation (30) is an alignment model which scores how well the output at step t depends on the input section related to the annotation vector $\mathrm{l_j}$. The function f in (30) measures the alignment between the output and the input. A simple candidate for implementing function f is an MLP which can be modeled as:

$$f(h_{t-1}, l_j) = W_2 \tanh(W_h h_{t-1} + W_l l_j + b_1) + b_2 \tag{31}$$

In which $\mathrm{W_2}$, $\mathrm{W_h}$, and $\mathrm{W_l}$ are weight matrices and b_1 and $\mathrm{b_2}$ are biases. All of these parameters can be trained jointly with other trainable model parameters.

Another version of the attention mechanism, called "Hard Attention", is also introduced by Xu et al. [10] in which at each step one of the attention weights is equal to 1 and the rest are equal to zero.

5.2 Extensions

Vaswani et al. [61] showed that the attention mechanism not only can be used instead of convolutional and recurrent layers in the network architecture, but also outperforms their functionality and decreases the computation complexity of the network training. Vaswani et al. [61] proposed a novel neural architecture in which all of the convolutional and recurrent layers of the networks are substituted with attention layers.

Attention mechanism is also used in other tasks. You et al. [62] proposed a semantic attention in image captioning. Lu et al. [22] also proposed an adaptive version of the attention mechanism in image captioning. Gao et al. [63] proposed an encoder-decoder based model using the attention mechanism and introduced a novel approach to train the attention weights to decrease the semantic gap of the generated caption.

Since the attention mechanism enables the models to focus more on a part of the input while generating the output, it can be used as a selector between multiple information sources. Wu et al. [64] proposed a novel encoder-decoder based model for video captioning which is used temporal features, audio features, motion features and semantic label features. The proposed attention-based hierarchical multi-modal fusion model (HATT) exploits the complementariness of multi-modal features. The model consists of three different attention layers: (1) low-level attention which deals with temporal, motion and audio features, (2) high-level attention which deals with semantic labels, and (3) sequential attention which aggregates the information from the other two layers of attention.

In the proposed model two different types of the attention mechanism is proposed:

1. Intra-modality attention: the soft-attention applied on a feature set V containing features from a single modality.
2. Inter-modality attention: the soft-attention applied on a feature set V containing features from multiple modalities.

Figure 15 displays the structure of different attention layers proposed by Wu et al. [64]. In the low-level attention layer, an intra-modality attention is applied on the temporal features first to generate the context vector C_t^{tem}. Then an inter-modality attention selects between the temporal and the motion features generating C_t^{mt}. In parallel, an intra-attention modality is applied on the audio features, which generates the context vector C_t^{aud}. Finally, the average vector of the C_t^{mt} and the C_t^{aud} is computed and passed on. In the high-level attention layer, an intra-modality attention is used to generate the context vector of the semantic labels. Ultimately, in the sequential attention layer, an intra-modality attention is used to aggregate the context vectors from the low-level and the high-level attention layers.

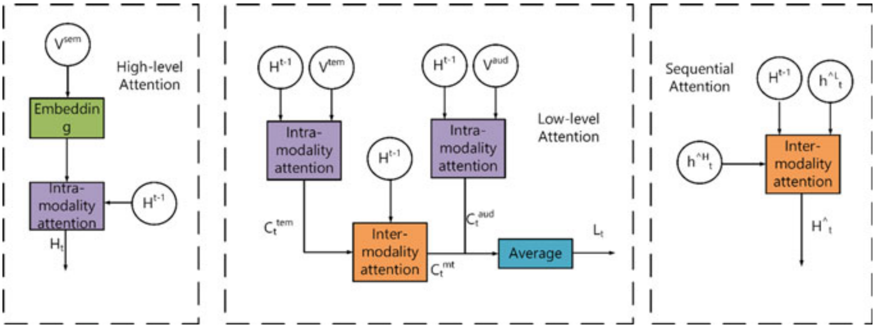


Fig. 15 Different attention layers

The proposed model by Wu et al. [64] employed the attention mechanism in a hierarchical structure to exploit both the long-term and the multi-modal feature dependencies from the input to generate fine-grained captions.

6 Future Work

The encoder-decoder based models are best fitted to the sophisticated tasks as an end-to-end solution and many different adaptations on them are accomplished. This provide a good starting point for discussion and further research.

Further studies on the encoder-decoder based models should investigate the following issues:

- Reinforcement learning
  One interesting way to enhance the performance of the encoder-decoder based models is to use reinforcement learning to train them.
- Long term dependencies
  A wide variety of ideas are proposed to cope with the long-term dependencies in the inputs. Current models are facing major problems to represent long sequences. We pointed out some of the ways to resolve this problem such as using LSTM cells, using the stacked structures for the decoders, treating the decoding phase as a regression problem rather than a classification one, and using the attention mechanism. Each of the introduced techniques can be improved and also new techniques can be added to this list.
- Applications
  As mentioned earlier, the encoder-decoder based models are widely used as an end-to-end solution for the sophisticated tasks. This is an interesting topic for the future work to employ the encoder-decoder based models for solving new problems.

7 Conclusion

In this chapter, we presented the baseline encoder-decoder model first proposed in machine translation and then extended to other applications. The main idea in the encoder-decoder framework is to split the process of generating a textual output describing the input into two subprocesses. A feature vector is first extracted by an encoder from the input. Then a decoder is used to generate the output step by step using the feature vector extracted by the encoder.

The structure of the encoder varies based on the input type. Whenever the input is a text or a sequence, an RNN is used as the encoder. When the input is an image, CNNs can be used to encode the input image. If that the input is a video a combination of

CNNs and RNNs is used to first extract features from all of the input frames and then model temporal dependencies between different frames. 3D-CNNs are also proposed to extract motion data from the input. More complex models to detect and localize events in the input video for dense video captioning are also discussed.

Extracting long-term dependencies and generating rich fine-grained sentences are the main problems of decoders in the encoder-decoder based models. LSTM cells are the simplest models that can cope with the problem of long-term dependencies, and are used in the proposed architectures for different tasks. Using stacked structures of RNNs is another approach to make deeper decoders. This allows the models to cope with the long-term dependency challenge and generate more detailed sentences. One of the most important problems of the stacked decoder structures is the problem of the vanishing gradients which can be solved by modifying the optimization problem of the decoder. Another approach to cope with the problem of long-term dependencies is using the attention mechanism, which is a technique to focus more on different portions of the input, while generating outputs at each step.

References

1. Lopez, A.: Statistical machine translation. ACM Comput. Surv. **40**(3), 8 (2008)
2. Cho, K., et al.: Learning phrase representations using RNN encoder-decoder for statistical machine translation (2014). http://arxiv.org/abs/1406.1078
3. Cho, K., Van Merriënboer, B., Bahdanau, D., Bengio, Y.: On the properties of neural machine translation: encoder-decoder approaches (2014). http://arxiv.org/abs/1409.1259
4. Venugopalan, S., Xu, H., Donahue, J., Rohrbach, M., Mooney, R., Saenko, K.: Translating videos to natural language using deep recurrent neural networks (2014). http://arxiv.org/abs/1412.4729
5. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems, pp. 3104–3112 (2014)
6. Gehring, J., Auli, M., Grangier, D., Dauphin, Y.N.: A convolutional encoder model for neural machine translation (2016). http://arxiv.org/abs/1611.02344
7. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
8. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate (2014). http://arxiv.org/abs/1409.0473
9. Luong, M.-T., Manning, C.D.: Stanford neural machine translation systems for spoken language domains. In: Proceedings of the International Workshop on Spoken Language Translation, pp. 76–79 (2015)
10. Xu, K., et al.: Show, attend and tell: neural image caption generation with visual attention. In: International Conference on Machine Learning, pp. 2048–2057 (2015)
11. Mi, H., Sankaran, B., Wang, Z., Ittycheriah, A.: Coverage embedding models for neural machine translation (2016). http://arxiv.org/abs/1605.03148
12. He, D., et al.: Dual learning for machine translation. In: Advances in Neural Information Processing Systems, pp. 820–828 (2016)
13. Tu, Z., Liu, Y., Lu, Z., Liu, X., Li, H.: Context gates for neural machine translation. Trans. Assoc. Comput. Linguist. **5**, 87–99 (2017)
14. Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: BLEU: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, pp. 311–318 (2002)

15. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
16. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). http://arxiv.org/abs/1409.1556
17. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L.: Imagenet: a large-scale hierarchical image database. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2009, pp. 248–255 (2009)
18. Karpathy, A., Fei-Fei, L.: Deep visual-semantic alignments for generating image descriptions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3128–3137 (2015)
19. Chen, L., et al.: SCA-CNN: spatial and channel-wise attention in convolutional networks for image captioning. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 6298–6306 (2017)
20. Pedersoli, M., Lucas, T., Schmid, C., Verbeek, J.: Areas of attention for image captioning. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1242–1250 (2017)
21. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
22. Lu, J., Xiong, C., Parikh, D., Socher, R.: Knowing when to look: adaptive attention via a visual sentinel for image captioning. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3242–3250 (2017)
23. Rennie, S.J., Marcheret, E., Mroueh, Y., Ross, J., Goel, V.: Self-critical sequence training for image captioning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7008–7024 (2017)
24. Anderson, P., et al.: Bottom-up and top-down attention for image captioning and VQA (2017). http://arxiv.org/abs/1707.07998
25. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2818–2826 (2016)
26. Zhang, L., et al.: Actor-critic sequence training for image captioning (2017). http://arxiv.org/abs/1706.09601
27. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift (2015). http://arxiv.org/abs/1502.03167
28. Vinyals, O., Toshev, A., Bengio, S., Erhan, D.: Show and tell: a neural image caption generator. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3156–3164 (2015)
29. Liu, S., Zhu, Z., Ye, N., Guadarrama, S., Murphy, K.: Improved image captioning via policy gradient optimization of spider. In: 2017 IEEE International Conference on Computer Vision (ICCV), pp. 873–881 (2017)
30. Yao, T., Pan, Y., Li, Y., Qiu, Z., Mei, T.: Boosting image captioning with attributes. In: 2017 IEEE International Conference on Computer Vision (ICCV), pp. 4904–4912 (2017)
31. Asadi, A., Safabakhsh, R.: A deep decoder structure based on word embedding regression for an encoder-decoder based model for image captioning. In: Submitted to Cognitive Computation (2019)
32. Lin, T.-Y., et al.: Microsoft COCO: Common Objects in Context. In: European Conference on Computer Vision, pp. 740–755 (2014)
33. Banerjee, S., Lavie, A.: METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pp. 65–72 (2005)
34. Vedantam, R., Lawrence Zitnick, C., Parikh, D.: Cider: consensus-based image description evaluation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4566–4575 (2015)

35. Lin, C.Y.: Rouge: a package for automatic evaluation of summaries. In: Proceedings of the Workshop on Text Summarization Branches Out (WAS 2004) (2004)
36. Majd, M., Safabakhsh, R.: Correlational convolutional LSTM for human action recognition. Neurocomputing (2019)
37. Majd, M., Safabakhsh, R.: A motion-aware ConvLSTM network for action recognition. Appl. Intell., 1–7 (2019)
38. Yao, L., et al.: Describing videos by exploiting temporal structure. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4507–4515 (2015)
39. Pan, Y., Mei, T., Yao, T., Li, H., Rui, Y.: Jointly modeling embedding and translation to bridge video and language. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4594–4602 (2016)
40. Li, Y., Yao, T., Pan, Y., Chao, H., Mei, T.: Jointly localizing and describing events for dense video captioning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7492–7500 (2018)
41. Krishna, R., Hata, K., Ren, F., Fei-Fei, L., Carlos Niebles, J.: Dense-captioning events in videos. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 706–715 (2017)
42. Escorcia, V., Heilbron, F.C., Niebles, J.C., Ghanem, B.: Daps: deep action proposals for action understanding. In: European Conference on Computer Vision, pp. 768–784 (2016)
43. Shen, Z., et al.: Weakly supervised dense video captioning. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 5159–5167 (2017)
44. Duan, X., Huang, W., Gan, C., Wang, J., Zhu, W., Huang, J.: Weakly supervised dense event captioning in videos. In: Advances in Neural Information Processing Systems, pp. 3063–3073 (2018)
45. Zhou, L., Zhou, Y., Corso, J.J., Socher, R., Xiong, C.: End-to-end dense video captioning with masked transformer. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 8739–8748 (2018)
46. Wang, J., Jiang, W., Ma, L., Liu, W., Xu, Y.: Bidirectional attentive fusion with context gating for dense video captioning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7190–7198 (2018)
47. Venugopalan, S., Rohrbach, M., Donahue, J., Mooney, R., Darrell, T., Saenko, K.: Sequence to sequence-video to text. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4534–4542 (2015)
48. Chen, D.L., Dolan, W.B.: Collecting highly parallel data for paraphrase evaluation. In: Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (HLT'11) (2011)
49. Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep Learning, vol. 1, p. 334. MIT press Cambridge (2016)
50. Luong, M.-T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation (2015). http://arxiv.org/abs/1508.04025
51. Wu, Y., et al.: Google's neural machine translation system: bridging the gap between human and machine translation (2016). http://arxiv.org/abs/1609.08144
52. Johnson, M., et al.: Google's multilingual neural machine translation system: enabling zero-shot translation. Trans. Assoc. Comput. Linguist. **5**, 339–351 (2017)
53. Luong, M.-T., Sutskever, I., Le, Q.V., Vinyals, O., Zaremba, W.: Addressing the rare word problem in neural machine translation (2014). http://arxiv.org/abs/1410.8206
54. Donahue, J., et al.: Long-term recurrent convolutional networks for visual recognition and description. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2625–2634 (2015)
55. Gu, J., Cai, J., Wang, G., Chens, T.: Stack-captioning: coarse-to-fine learning for image captioning. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
56. Pan, P., Xu, Z., Yang, Y., Wu, F., Zhuang, Y.: Hierarchical recurrent neural encoder for video representation with application to captioning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1029–1038 (2016)

57. Yu, H., Wang, J., Huang, Z., Yang, Y., Xu, W.: Video paragraph captioning using hierarchical recurrent neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4584–4593 (2016)
58. Chen, H., Ding, G., Lin, Z., Zhao, S., Han, J.: Show, observe and tell: attribute-driven attention model for image captioning. In: IJCAI International Joint Conference on Artificial Intelligence (2018)
59. Ding, G., Chen, M., Zhao, S., Chen, H., Han, J., Liu, Q.: Neural image caption generation with weighted training and reference. Cogn. Comput. (2018)
60. Wang, X., Chen, W., Wu, J., Wang, Y.-F., Yang Wang, W.: Video captioning via hierarchical reinforcement learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4213–4222 (2018)
61. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)
62. You, Q., Jin, H., Wang, Z., Fang, C., Luo, J.: Image captioning with semantic attention. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4651–4659 (2016)
63. Gao, L., Guo, Z., Zhang, H., Xu, X., Shen, H.T.: Video captioning with attention-based lstm and semantic consistency. IEEE Trans. Multimed. **19**(9), 2045–2055 (2017)
64. Wu, C., Wei, Y., Chu, X., Weichen, S., Su, F., Wang, L.: Hierarchical attention-based multimodal fusion for video captioning. Neurocomputing **315**, 362–370 (2018)

Deep Learning for Learning Graph Representations

Wenwu Zhu, Xin Wang and Peng Cui

Abstract Mining graph data has become a popular research topic in computer science and has been widely studied in both academia and industry given the increasing amount of network data in the recent years. However, the huge amount of network data has posed great challenges for efficient analysis. This motivates the advent of graph representation which maps the graph into a low-dimension vector space, keeping original graph structure and supporting graph inference. The investigation on efficient representation of a graph has profound theoretical significance and important realistic meaning, we therefore introduce some basic ideas in graph representation/network embedding as well as some representative models in this chapter.

Keywords Deep learning · Graph representation · Network embedding

1 Introduction

Many real-world systems, such as Facebook/Twitter social systems, DBLP author-citation systems and roadmap transportation systems etc., can be formulated in the form of graphs or networks, making analyzing these systems equivalent to mining their corresponding graphs or networks. Literature on mining graphs or networks has two names: graph representation and network embedding. We remark that *graph* and *network* all refer to the same kind of structure, although each of them may have its own terminology, e.g., a *vertice* and an *edge* in a graph v.s. a *node* and a *link* in a network. Therefore we will exchangeably use graph representation and network embedding without further explanations in the remainder of this chapter. The core

W. Zhu (✉) · X. Wang · P. Cui
Tsinghua University, Beijing, China
e-mail: wwzhu@tsinghua.edu.cn

X. Wang
e-mail: xin_wang@tsinghua.edu.cn

P. Cui
e-mail: cuip@tsinghua.edu.cn

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_6

of mining graphs/networks relies heavily on properly representing graphs/networks, making representation learning on graphs/networks a fundamental research problem in both academia and industry. Traditional representation approaches represent graphs directly based on their topologies, resulting in many issues including sparseness, high computational complexities etc., which actuates the advent of machine learning based methods that explore the latent representations capable of capturing extra information in addition to topological structures for graphs in vector space. As such, the ability to find "good" latent representations for graphs plays an important role in accurate graph representations. However, learning network representations faces challenges as follows:

1. **High non-linearity**. As is claimed by Luo et al. [43], the network has highly non-linear underlying structure. Accordingly, it is a rather challenging work to design a proper model to capture the *highly non-linear* structure.
2. **Structure-preserving**. With the aim of supporting network analysis applications, preserving the network structure is required for network embedding. However, the underlying structure of the network is quite *complex* [55]. In that the similarity of vertexes depends on both the local and global network structure, it is a tough problem to preserve the local and global structure simultaneously.
3. **Property-preserving**. Real-world networks are normally very complex, their formation and evolution are accompanied with various properties such as uncertainties and dynamics. Capturing these properties in graph representation is of significant importance and poses great challenges.
4. **Sparsity**. Real-world networks are often too *sparse* to provide adequate observed links for utilization, consequently causing unsatisfactory performances [50].

Many network embedding methods have been put forward, which adopt shallow models like IsoMAP [62], Laplacian Eigenmaps (LE) [4] and Line [61]. However, on account of the limited representation ability [6], it is challenging for them to capture the highly nonlinear structure of the networks [63]. As [76] stated, although some methods adopt kernel techniques [68], they still belong to shallow models, incapable of capturing the highly non-linear structure well. On the other hand, the success of deep learning in handling non-linearity brings us great opportunities for accurate representations in latent vector space. One natural question is that can we utilize deep learning to boost the performance of graph representation learning? The answer is yes, and we will discuss some recent advances in combining deep learning techniques with graph representation learning in this chapter. Our discussions fall in two categories of approaches: deep structure-oriented approaches and deep property-oriented approaches. For structure-oriented approaches, we include three methods as follows.

- Structural deep network embedding (SDNE) [69] that focuses on preserving high order proximity.
- Deep recursive network embedding (DRNE) [66] that focuses on preserving global structure.
- Deep hyper-network embedding (DHNE) [65] that focuses on preserving hyper structure.

For property-oriented approaches, we discuss:

- Deep variational network embedding (DVNE) [73] that focuses on the uncertainty property.
- Deeply transformed high-order Laplacian Gaussian process (DepthLGP) based network embedding [44] that focuses on the dynamic (i.e., out-of-sample) property.

Deep Structure-Oriented Methods

2 High Order Proximity Preserving Network Embedding

Deep learning, as a powerful tool capable of learning complex structures of the data [6] through efficient representation, has been widely adopted to tackle a large number of tasks related to image [38], audio [29] and text [58] etc. To preserve the high order proximity as well as capture the **highly non-linear** structure, Wang et al. [69] propose to learn vertex representations for networks by resorting to autoencoder, motivated by the recent success of deep neural networks. Concretely, the authors design a multi-layer architecture containing multiple non-linear functions, which maps the data into a highly non-linear latent space, thus is able to capture the highly non-linear network structure.

So as to resolve the **structure-preserving** and **sparsity** problems in the deep models, the authors further put forward a method to jointly mine the first-order and second-order proximity [61] during the learning process, where the former captures the local network structure, only characterizing the local pairwise similarity between the vertexes linked by edges. Nonetheless, many legitimate links are missing due to the sparsity of the network. Consequently, the first-order proximity alone cannot represent the network structure sufficiently. Therefore, the authors further advance the second-order proximity, the indication of the similarity among the vertexes' neighborhood structures, to characterize the global structure of networks. With the first-order and second-order proximity adopted simultaneously, the model can capture both the local and global network structures well respectively. The authors also propose a semi-supervised architecture to preserve both the local and global network structure in the deep model, where the first-order proximity is exploited as the supervised information by the supervised component exploits, preserving the local one, while the second-order proximity is reconstructed by the unsupervised component, preserving the global one. Moreover, as is illustrated in Fig. 1, there are much more pairs of vertexes having second-order proximity than first-order proximity. Hence, in the light of characterizing the network structure, importing second-order proximity can provide much more information. In general, for purpose of preserving the network structure, SDNE is capable of mapping the data to a highly non-linear latent space while it is also robust to sparse networks. To our best knowledge, SDNE is among the first to adopt deep learning structures for network representation learning.

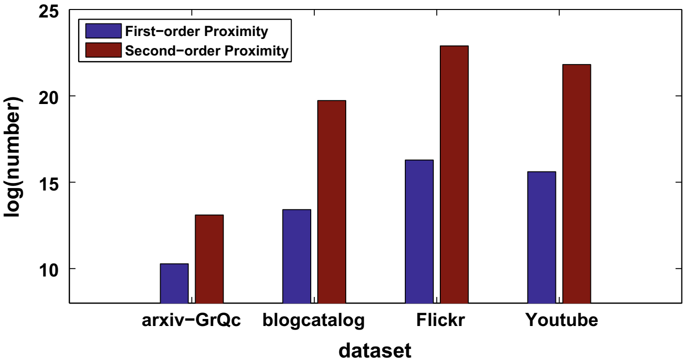


Fig. 1 The number of pairs of vertexes which have first-order and second-order proximity in different datasets, figure from [69]

2.1 Problem Definition

Definition 2.1 (*Graph*) $G = (V, E)$ represents a graph, where $V = \{v_1, ..., v_n\}$ stands for n vertexes and $E = \{e_{i,j}\}_{i,j=1}^{n}$ stands for the edges. Each edge $e_{i,j}$ is associated with a weight $s_{i,j} \geq 0$.[1] For v_i and v_j without being linked by any edge, $s_{i,j} = 0$. Otherwise, $s_{i,j} = 1$ for unweighted graph and $s_{i,j} > 0$ for weighted graph.

The goal of network embedding is mapping the graph data into a lower-dimensional latent space. Specifically, each vertex is mapped to a low-dimensional vector so that the network computation can be directly done in that latent space. As mentioned before, preserving both local and global structure is essential. First, the first-order proximity able to characterize the local network structure, is defined as follows.

Definition 2.2 (*First-Order Proximity*) The first-order proximity represents the pair-wise proximity between vertexes. For a vertex pair, first-order proximity between v_i and v_j is positive if $s_{i,j} > 0$ and 0 otherwise.

Spontaneously, network embedding is requisite to preserve the first-order proximity for the reason that it means that two vertexes linked by an observed edge in real-world networks are always similar. For instance, if a paper is cited by another, they are supposed to have some common topics. Nonetheless, real-world datasets often have such high sparsity that only a small portion is the observed links. Many vertexes with similarity are not linked by any edges in the networks. Accordingly, it is not sufficient to only capture the first-order proximity, which is why the second-order proximity is introduced as follows to characterize the global network structure.

[1] Negative links exist in signed network, but only non-negative links are considered here.

Definition 2.3 (*Second-Order Proximity*) The second-order proximity of a vertex pair represents the proximity of the pair's neighborhood structure. Let $\mathcal{N}_u = \{s_{u,1}, ..., s_{u,|V|}\}$ stand for the first-order proximity between v_u and other vertexes. Second-order proximity is then decided by the similarity of $\mathcal{N}_u$ and $\mathcal{N}_v$.

Intuitively, the second-order proximity presumes two vertexes to be similar if they share many common neighbors. In many fields, such an assumption has been proved reasonable [18, 35]. For instance, if two linguistics words always have similar contexts, they will usually be similar [18]. People sharing many common friends tend to be friends [35]. It has been demonstrated that the second-order proximity is a good metric for defining the similarity between vertex pairs even without being linked by edges [42], which can also highly improve the richness of vertex relationship consequently. Thus, taking the second-order proximity into consideration enables the model to capture the global network structure and relieve the sparsity problem as well.

To preserve both the local and global structure when in network embedding scenarios, we now focus on the problem of how to integrate the first-order and second-order proximity simultaneously, the definition of which is as follows.

Definition 2.4 (*Network Embedding*) Given a graph $G = (V, E)$, the goal of network embedding is learning a mapping function $f : v_i \longmapsto \mathbf{y_i} \in \mathbb{R}^d$, where $d \ll |V|$. The target of the function is to enable the similarity between $\mathbf{y_i}$ and $\mathbf{y_j}$ to preserve the *first-order* and *second-order* proximity of v_i and v_j explicitly.

2.2 The SDNE Model

This section discusses the semi-supervised SDNE model for network embedding, the framework of which is illustrated in Fig. 2. Specifically, for purpose of characterizing the highly non-linear network structure, the authors put forward a deep architecture containing numerous non-linear mapping functions to transform the input into a highly non-linear latent space. Moreover, for purpose of exploiting both the first-order and second-order proximity, a semi-supervised model is adopted, aiming to resolve the problems of structure-preserving and sparsity. We are able to obtain the neighborhood of each vertex. Hence, to preserve the second-order proximity by the method of reconstructing the neighborhood structure of every vertex, the authors project the unsupervised component. At the same time, for a small portion of vertex pairs, obtaining their pairwise similarities (i.e. the first-order proximity) is also possible. Thus, the supervised component is also adopted to exploit the first-order proximity as the supervised information for refining the latent representations. By optimizing these two types of proximity jointly in the semi-supervised model proposed, *SDNE* is capable of preserving the highly-nonlinear local and global network structure well and is also robust when dealing with sparse networks.

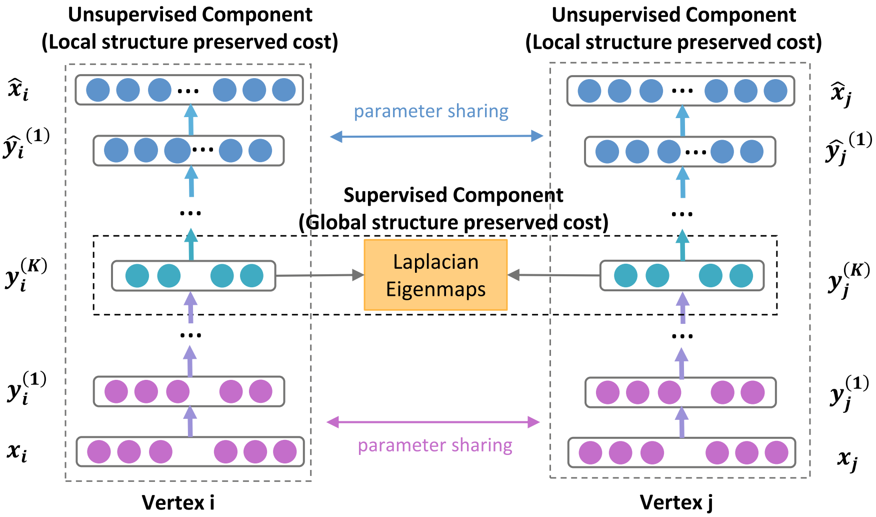


Fig. 2 Framework of the *SDNE* model, figure from [69]

2.2.1 Loss Functions

We give definition of some notations and terms, before defining the loss functions, to be used later in Table 1. Note that ˆ symbol above the parameters stands for decoder parameters.

To begin with, we describe how the second-order proximity is exploited by the unsupervised component in order to preserve the global network structure.

The second-order proximity refers to the similarity of the neighborhood structure of a vertex pair. Therefore, to capture it properly, we need to consider the neighborhood of each vertex when modeling. For a network $G = (V, E)$, its adjacency matrix S containing n instances $\mathbf{s}_1, ..., \mathbf{s}_n$ can be easily obtained. For each

Table 1 Terms and notations

Symbol	Definition
n	Number of vertexes
K	Number of layers
$S = \{\mathbf{s}_1, ..., \mathbf{s}_n\}$	The adjacency matrix for the network
$X = \{\mathbf{x}_i\}_{i=1}^n, \hat{X} = \{\hat{\mathbf{x}}_i\}_{i=1}^n$	The input data and reconstructed data
$Y^{(k)} = \{\mathbf{y}_i^{(k)}\}_{i=1}^n$	The k-th layer hidden representations
$W^{(k)}, \hat{W}^{(k)}$	The k-th layer weight matrix
$\mathbf{b}^{(\mathbf{k})}, \hat{\mathbf{b}}^{(k)}$	The k-th layer biases
$\theta = \{W^{(k)}, \hat{W}^{(k)}, \mathbf{b}^{(k)}, \hat{\mathbf{b}}^{(k)}\}$	The overall parameters

instance $\mathbf{s}_i = \{s_{i,j}\}_{j=1}^{n}$, $s_{i,j} > 0$ iff there exists a link between v_i and v_j. Therefore, $\mathbf{s}_i$ represents the neighborhood structure of the vertex v_i, i.e., S involves the information of each vertex's neighborhood structure. Based on S, conventional deep autoencoder [54] is extended for purpose of preserving the second-order proximity.

We briefly review the key ideas of deep autoencoder to be self-contained. A deep autoencoder is an unsupervised model consisting of an encoder and decoder. The two parts both consist of numerous non-linear functions, while the encoder maps the input data to the representation space and the decoder maps the representation space to reconstruction space. The hidden representations for each layer are defined as follows, given the input $\mathbf{x}_i$.[2]

$$\begin{aligned} \mathbf{y}_i^{(1)} &= \sigma(W^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)}) \\ \mathbf{y}_i^{(k)} &= \sigma(W^{(k)}\mathbf{y}_i^{(k-1)} + \mathbf{b}^{(k)}), k = 2, ..., K. \end{aligned} \tag{1}$$

With $\mathbf{y}_i^{(K)}$ obtained, the output $\hat{\mathbf{x}}_i$ can be obtained through the reversion the encoder's computation process. The goal of an autoencoder is minimizing the reconstruction error between input and output, to achieve which the following loss function can be defined.

$$\mathcal{L} = \sum_{i=1}^{n} \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|_2^2. \tag{2}$$

As proven by [54], the reconstruction formula can smoothly characterize the data manifolds and thus preserve it implicitly though not explicitly. Consider this case: if the adjacency matrix S is inputted into the autoencoder, i.e., $\mathbf{x}_i = \mathbf{s}_i$, the reconstruction process will output the similar latent representations for the vertexes with similar neighborhood structures, as each instance $\mathbf{s}_i$ captures the neighborhood structure of the vertex v_i.

However, owing to some specific characteristics of the networks, such a reconstruction process cannot fit our problem straightforward. In the networks, some links can be observed, while many other legitimate links cannot. It suggests that while the links among vertexes do indicate their similarity, having no links does not necessarily indicate dissimilarity between the vertexes. In addition, the number of 0 elements in S is far more than that of non-zero elements due to the sparsity of networks. Thus, by directly inputting S to the conventional autoencoder, there is a tendency of reconstructing the 0 elements in S, which does not fit out expectation. Therefore, with the help of the revised objective function as follows, SDNE imposes more penalty to the reconstructing error for the non-zero elements than that for 0 elements:

$$\begin{aligned} \mathcal{L}_{2nd} &= \sum_{i=1}^{n} \|(\hat{\mathbf{x}}_i - \mathbf{x}_i) \odot \mathbf{b_i}\|_2^2 \\ &= \|(\hat{X} - X) \odot B\|_F^2, \end{aligned} \tag{3}$$

[2] Here the authors use sigmoid function $\sigma(x) = \frac{1}{1+exp(-x)}$ as the non-linear activation function.

where $\odot$ stands for the Hadamard product, $\mathbf{b_i} = \{b_{i,j}\}_{j=1}^{n}$. If $s_{i,j} = 0$, $b_{i,j} = 1$, else $b_{i,j} = \beta > 1$. Now by inputting S to the revised deep autoencoder, vertexes with alike neighborhood structure will have close representations in the latent space, which is guaranteed by the reconstruction formula. In another word, reconstructing the second-order proximity among vertexes enables the unsupervised component of SDNE to preserve the global network structure.

As explained above, preserving the global and local network structure are both essential in this task. For purpose of representing the local network structure, the authors use first-order proximity, the supervised information for constraining the similarity among the latent representations of vertex pairs. Hence, the supervised component is designed to exploit this first-order proximity. The following definition of loss function is designed for this target.[3]

$$\begin{aligned}\mathcal{L}_{1st} &= \sum_{i,j=1}^{n} s_{i,j} \|\mathbf{y}_i^{(K)} - \mathbf{y}_j^{(K)}\|_2^2 \\ &= \sum_{i,j=1}^{n} s_{i,j} \|\mathbf{y}_i - \mathbf{y}_j\|_2^2.\end{aligned} \tag{4}$$

A penalty is incurred by this objective function when similar vertexes are mapped far away in the latent representation space, which borrows the idea of Laplacian Eigenmaps [4]. Other works on social networks [34] also use the similar idea. SDNE differs from these methods on the fact that it incorporates this idea into the deep model to embed the linked-by-edge vertexes close to each other in the latent representation space, preserving the first-order proximity consequently.

For purpose of preserving the first-order and second-order proximity simultaneously, Eqs. (4) and (3) is combined by SDNE through jointly minimizing the objective functions as follows.

$$\begin{aligned}\mathcal{L}_{mix} &= \mathcal{L}_{2nd} + \alpha\mathcal{L}_{1st} + \nu\mathcal{L}_{reg} \\ &= \|(\hat{X} - X) \odot B\|_F^2 + \alpha \sum_{i,j=1}^{n} s_{i,j} \|\mathbf{y}_i - \mathbf{y}_j\|_2^2 + \nu\mathcal{L}_{reg},\end{aligned} \tag{5}$$

where $\mathcal{L}_{reg}$ stands for an $\mathcal{L}2$-norm regularizer term as follows, aiming to avoid overfitting:

$$\mathcal{L}_{reg} = \frac{1}{2} \sum_{k=1}^{K} (\|W^{(k)}\|_F^2 + \|\hat{W}^{(k)}\|_F^2).$$

[3] To simplify the notations, network representations $Y^{(K)} = \{\mathbf{y}_i^{(K)}\}_{i=1}^{n}$ are denoted as $Y = \{\mathbf{y}_i\}_{i=1}^{n}$ by the authors.

2.2.2 Optimization

For purpose of optimizing the model mentioned above, we minimize $\mathcal{L}_{mix}$ as a function of θ. Specifically, the critical step is computing the partial derivative of $\partial\mathcal{L}_{mix}/\partial\hat{W}^{(k)}$ and $\partial\mathcal{L}_{mix}/\partial W^{(k)}$ with the following detailed mathematical form:

$$\begin{aligned}\frac{\partial\mathcal{L}_{mix}}{\partial\hat{W}^{(k)}} &= \frac{\partial\mathcal{L}_{2nd}}{\partial\hat{W}^{(k)}} + \nu\frac{\partial\mathcal{L}_{reg}}{\partial\hat{W}^{(k)}} \\ \frac{\partial\mathcal{L}_{mix}}{\partial W^{(k)}} &= \frac{\partial\mathcal{L}_{2nd}}{\partial W^{(k)}} + \alpha\frac{\partial\mathcal{L}_{1st}}{\partial W^{(k)}} + \nu\frac{\partial\mathcal{L}_{reg}}{\partial W^{(k)}}, k = 1, ..., K.\end{aligned} \tag{6}$$

First, we focus on $\partial\mathcal{L}_{2nd}/\partial\hat{W}^{(K)}$, which can be rephrased as follows.

$$\frac{\partial\mathcal{L}_{2nd}}{\partial\hat{W}^{(K)}} = \frac{\partial\mathcal{L}_{2nd}}{\partial\hat{X}} \cdot \frac{\partial\hat{X}}{\partial\hat{W}^{(K)}}. \tag{7}$$

In light of Eq. (3), for the first term we have

$$\frac{\partial\mathcal{L}_{2nd}}{\partial\hat{X}} = 2(\hat{X} - X) \odot B. \tag{8}$$

The computation of the second term $\partial\hat{X}/\partial\hat{W}$ is simple because $\hat{X} = \sigma(\hat{Y}^{(K-1)}\hat{W}^{(K)} + \hat{b}^{(K)})$, with which $\partial\mathcal{L}_{2nd}/\partial\hat{W}^{(K)}$ is available. Through back-propagation method, we can iteratively acquire $\partial\mathcal{L}_{2nd}/\partial\hat{W}^{(k)}$, $k = 1, ...K - 1$ and $\partial\mathcal{L}_{2nd}/\partial W^{(k)}$, $k = 1, ...K$.

Next, we move on to the partial derivative of $\partial\mathcal{L}_{1st}/\partial W^{(k)}$. The loss function of $\mathcal{L}_{1st}$ can be rephrased as follows.

$$\mathcal{L}_{1st} = \sum_{i,j=1}^{n} s_{i,j}\|\mathbf{y}_i - \mathbf{y}_j\|_2^2 = 2tr(Y^T LY), \tag{9}$$

where $L = D - S$, $D \in \mathbb{R}^{n\times n}$ stands for the diagonal matrix where $D_{i,i} = \sum_j s_{i,j}$.

Next, we center upon the computation of $\partial\mathcal{L}_{1st}/\partial W^{(K)}$ first:

$$\frac{\partial\mathcal{L}_{1st}}{\partial W^{(K)}} = \frac{\partial\mathcal{L}_{1st}}{\partial Y} \cdot \frac{\partial Y}{\partial W^{(K)}}. \tag{10}$$

Because $Y = \sigma(Y^{(K-1)}W^{(K)} + b^{(K)})$, the computation of the second term $\partial Y/\partial W^{(K)}$ is simple. For $\partial\mathcal{L}_{1st}/\partial Y$, we hold

$$\frac{\partial\mathcal{L}_{1st}}{\partial Y} = 2(L + L^T) \cdot Y. \tag{11}$$

Likewise, the calculation of partial derivative of $\mathcal{L}_{1st}$ can be finally finished through back-propagation.

All the partial derivatives of the parameters have been acquired now. After implementing parameter initialization, we can optimize the deep model proposed above with stochastic gradient descent. It is worth mentioning that owing to its high nonlinearity, the model may fall into many local optimum in the parameter space. Hence, the authors adopt Deep Belief Network as a method of pretraining at first [30] to find a good region in parameter space, which has been proved to be an fundamental way of initialization for deep learning architectures [24]. Algorithm 1 presents the complete algorithm.

2.3 Analysis and Discussions on SDNE

New Vertexes. Learning representations for newly arrived vertexes is a practical issue for network embedding. If we know the connections of a new vertex v_k to the existing vertexes, its adjacency vector $\mathbf{x} = \{s_{1,k}, ..., s_{n,k}\}$ is easy to obtained, where $s_{i,k}$ indicates the similarity between the new vertex v_k and the existing v_i. Then $\mathbf{x}$ can be simply fed into the deep model, after which we can calculate the representations for v_k with the trained parameters θ. For such a process, the time complexity is $O(1)$. Nonetheless, SDNE fails when there are no connections between existing vertexes and v_k in the network.

Algorithm 1 Training Algorithm for *SDNE*

```
Require: the network $G = (V, E)$ with adjacency matrix $S$, the parameters $\nu$ and $\alpha$
Ensure: Representations Y of the network and updated Parameters: $\theta$
Pretrain the model with deep belief network, obtaining the parameters $\theta = \{\theta^{(1)}, ..., \theta^{(K)}\}$
$X = S$
repeat
    Apply Eq. (1) to calculate $\hat{X}$ and $Y = Y^K$ given $X$ and $\theta$.
    $\mathcal{L}_{mix}(X; \theta) = \|(\hat{X} - X) \odot B\|_F^2 + 2\alpha tr(Y^T LY) + \nu\mathcal{L}_{reg}$.
    Utilize $\partial L_{mix}/\partial\theta$ to back-propagate throughout the entire network based on Eq. (6) to calculate the updated parameters of $\theta$.
until convergence
Return the network representations $Y = Y^{(K)}$
```

Training Complexity. The complexity of SDNE is $O(ncdI)$, where n stands for the number of vertexes, c represents the average degree of the network, d stands for the maximum dimension of the hidden layer, and I represents the number of iterations. d is often related to the dimension of the embedding vectors but not n, while I is also independent of n. In real-world applications, the parameter c can be regarded as a constant. For instance, the maximum number of a user's friends is always bounded [63] in a social network. Meanwhile, $c = k$ in a top-k similarity graph. Thus, cdI is also independent of n. So the total training complexity is actually linear to n .

3 Global Structure Preserving Network Embedding

As is discussed, one fundamental problem of network embedding is how to preserve the vertex similarity in an embedding space, i.e., two vertexes should have the similar embedding vectors if they have similar local structures in the original network. To quantify the similarity among vertexes in a network, the most common one among multiple methods is *structural equivalence* [41], where two vertexes sharing lots of common network neighbors are considered structurally equivalent. Besides, preserving structural equivalence through high-order proximities [61, 69] is the aim of most previous work on network embedding, where network neighbors are extended to high-order neighbors, e.g., direct neighbors and neighbors-of-neighbors, etc.

However, vertexes without any common neighbors can also occupy similar positions or play similar roles in many cases. For instance, two mothers share the same pattern of connecting with several children and one husband (the father). The two mothers do share similar positions or roles although they are not structurally equivalent if they do not share any relatives. These cases lead to an extended definition of vertex similarity known as *regular equivalence*: two regularly equivalent vertexes have network neighbors which are themselves similar (i.e., regularly equivalent) [52]. As neighbor relationships in a network can be defined recursively, we remark that regularly equivalence is able to reflect the global structure of a network. Besides, regular equivalence is, apparently, a relaxation of structural equivalence. Structural equivalence promises regular equivalence, while the reverse direction does not hold. Comparatively, regular equivalence is more capable and flexible of covering a wider range of network applications with relation to node importance or structural roles, but is largely ignored by previous work on network embedding.

For purpose of preserving global structure and regular equivalence in network embedding, i.e., two nodes of regularly equivalence should have similar embeddings, a simple way is to explicitly compute the regular equivalence of all vertex pairs and preserve the similarities of corresponding embeddings of nodes to approximate their regular equivalence. Nevertheless, due to the high complexity in computing regular equivalence in large-scale networks, this idea is infeasible. Another way is to replace regular equivalence with simpler graph theoretic metrics, such as centrality measures. Although many centrality measures have been proposed to characterize the importance and role of a vertex, it is still difficult to learn general and task-independent node embeddings because one centrality can only capture a specific aspect of network role. In addition, some centrality measures, e.g., betweeness centrality, also bear high computational complexity. Thus, how to efficiently and effectively preserve regular equivalence in network embedding is still an open problem.

Fortunately, the recursive nature in the definition of regular equivalence enlightens Tu et al. [66] to learn network embedding in a recursive way, i.e., the embedding of one node can be aggregated by its neighbors' embeddings. In one recursive step (Fig. 3), if nodes 7 and 8, 4 and 6, 3 and 5 are regularly equivalent and thus have similar embeddings already, then nodes 1 and 2 would also have similar embeddings, resulting in their regular equivalence. It is this idea that inspires the design of the

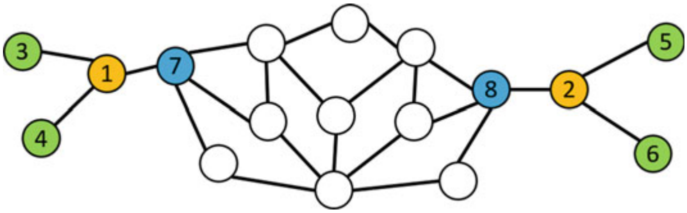


Fig. 3 A simple graph to illustrate the rationality of why recursive embedding can preserve regular equivalence. The regularly equivalent nodes share the same color, figure from [66]

Deep Recursive Network Embedding (DRNE) model [66]. In specific, the neighbors of a node are transformed into an ordered sequence and a layer normalized LSTM (Long Short Term Memory networks) [31] is proposed to aggregate the embeddings of neighbors into the embedding of the target node in a non-linear way.

3.1 Preliminaries and Definitions

In this section, we discuss the Deep Recursive Network Embedding (DRNE) model whose framework is demonstrated in Fig. 4. Taking node 0 in Fig. 4 as an example, we sample three nodes 1, 2, 3 from its neighborhoods and sort them by degree as a sequence (3, 1, 2). We use the embeddings of the neighborhoods sequence $\mathbf{X}_3$, $\mathbf{X}_1$, $\mathbf{X}_2$ as input, aggregating them by a layer normalized LSTM to get the assembled representation h_T. By reconstructing the embedding $\mathbf{X}_0$ of node 0 with the aggregated representation h_T, the embedding $\mathbf{X}_0$ can preserve the local neighborhood structure. On the other hand, we use the degree d_0 as weak supervision information of centrality and put the aggregated representation h_T into a multilayer perceptron (MLP) to approximate degree d_0. The same process is conducted for each other node. When we

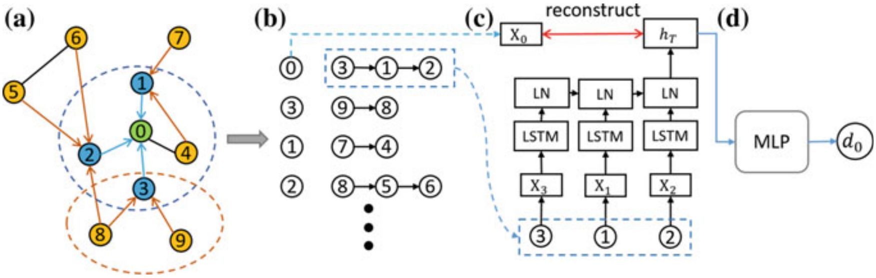


Fig. 4 Deep Recursive Network Embedding (DRNE). **a** Sampling neighborhoods. **b** Sorting neighborhoods by their degrees. **c** Layer-normalized LSTM to aggregate embeddings of neighboring nodes into the embedding of the target node. X_i is the embedding of node i and LN means layer normalization. **d** A Weakly guided regularizer. Figure from [66]

update the embedding of the neighborhoods $\mathbf{X}_3, \mathbf{X}_1, \mathbf{X}_2$, it will affect the embeddings $\mathbf{X}_0$. Repeating this procedure by updating the embeddings iteratively, the embeddings $\mathbf{X}_0$ can contain structural information of the whole network.

Given a network $G = (V, E)$, where V stands for the set of nodes and $E \in V \times V$ edges. For a node $v \in V$, $\mathcal{N}(v) = \{u|(v, u) \in E\}$ represents the set of its neighborhoods. The learned embeddings are defined as $\mathbf{X} \in \mathbb{R}^{|V|\times k}$ where k is the dimension and $\mathbf{X}_v \in \mathbb{R}^k$ is the embedding of node v. The degree of node v is defined as $d_v = |\mathcal{N}(v)|$ and function $I(x) = 1$ if $x \geq 0$ otherwise 0. The formal definition of structural equivalence and regular equivalence is given as follows.

Definition 3.1 (*Structural Equivalence*) We denote $s(u) = s(v)$ if nodes u and v are structurally equivalent. Then $s(u) = s(v)$ if and only if $\mathcal{N}(u) = \mathcal{N}(v)$.

Definition 3.2 (*Regular Equivalence*) We denote $r(u) = r(v)$ if nodes u and v are regularly equivalent. Then $r(u) = r(v)$ if and only if $\{r(i)|i \in \mathcal{N}(u)\} = \{r(j)|j \in \mathcal{N}(u)\}$.

3.2 The DRNE Model

3.2.1 Recursive Embedding

In light of Definition 3.2, DRNE learns the embeddings of nodes recursively: the embedding of a target node can be approximated by aggregating the embeddings of its neighbors, So we can use the following loss function:

$$\mathcal{L}_1 = \sum_{v\in V} ||\mathbf{X}_v - Agg(\{\mathbf{X}_u|u \in \mathcal{N}(v)\})||_F^2, \tag{12}$$

where Agg is the function of aggregation. In each recursive step, the local structure of the neighbors of the target node can be preserved by its learned embedding. Therefore, the learned node embeddings can incorporate the structural information in a global sense by updating the learned representations iteratively, which consists with the definition of regular equivalence.

As for the aggregating function, DRNE utilizes the layer normalized Long Short-Term Memory(ln-LSTM) [3] due to the highly nonlinearity of the underlying structures of many real networks [43]. LSTM is an effective model for modeling sequences, as is known to all. Nonetheless, in networks, the neighbors of a node have no natural ordering. Here the degree of nodes is adopted as the criterion of sorting neighbors in an ordered sequence for the reason that taking degree as measure for neighbor ordering is the most efficient and that degree is a crucial part in many graph-theoretic measures, notably those relating to structural roles, e.g. Katz [47] and PageRank [49].

Suppose $\{X_1, X_2, ..., X_t, ..., X_T\}$ are the embeddings of the ordered neighbors. At each time step t, the hidden state h_t is a function of its previous hidden state h_{t-1} and input embedding X_t, i.e., $h_t = LSTMCell(h_{t-1}, X_t)$. The information of hidden representation h_t will become increasingly abundant while the embedding sequence is processed by LSTM Cell recursively from 1 to T. Therefore, h_T can be treated as the aggregation of the representation from neighbors. In addition, LSTM with gating mechanisms is effective in learning long-distance correlations in long sequences. In the structure of LSTM, the input gate along with old memory decides what new information to be stored in memory, the forget gate decides what information to be thrown away from the memory and output gate decides what to output based on the memory. Specifically, *LSTMCell*, the LSTM transition equation, can be written as follows.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \cdot [\mathbf{h}_{t-1}, \mathbf{X}_t] + \mathbf{b}_f), \tag{13}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \cdot [\mathbf{h}_{t-1}, \mathbf{X}_t] + \mathbf{b}_i), \tag{14}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \cdot [\mathbf{h}_{t-1}, \mathbf{X}_t] + \mathbf{b}_o), \tag{15}$$

$$\tilde{\mathbf{C}}_t = tanh(\mathbf{W}_C \cdot [\mathbf{h}_{t-1}, \mathbf{X}_t] + \mathbf{b}_C), \tag{16}$$

$$\mathbf{C}_t = \mathbf{f}_t * \mathbf{C}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{C}}_t, \tag{17}$$

$$\mathbf{h}_t = \mathbf{o}_t * tanh(\mathbf{C}_t), \tag{18}$$

where σ stands for the sigmoid function, $*$ and $\cdot$ represent element-wise product and matrix product respectively, C_t represents the cell state, $\mathbf{i}_t$, $\mathbf{f}_t$ and $\mathbf{o}_t$ are input gate, forget gate and output gate respectively. $\mathbf{W}_*$ and $\mathbf{b}_*$ are the parameters to be learned.

Moreover, the Layer Normalization [3] is adopted in DRNE for purpose of avoiding the problems of exploding or vanishing gradients [31] when long sequences are taken as input. The layer normalized LSTM makes re-scaling all of the summed input invariant, resulting in much more stable dynamics. In particular, with the extra normalization as follows, it re-centers and re-scales the cell state C_t after Eq. (17).

$$C'_t = \frac{g}{\Sigma_t} * (C_t - \mu_t), \tag{19}$$

where $\mu_t = 1/k \sum_{i=1}^{k} C_{ti}$ and $\Sigma_t = \sqrt{1/k \sum_{i=1}^{k} (C_{ti} - \mu_t)^2}$ are the mean and variance of C_t, and g is the gain parameter scaling the normalized activation.

3.2.2 Regularization

Without any other constraints, $\mathcal{L}_1$ defined in Eq. (12) represents the recursive embedding process according to Definition 3.2. Its expressive power is so strong that we can obtain multiple solutions to satisfy the recursive process defined above. The model take a risk of degenerating to a trivial solution: all the embeddings become 0. For purpose of avoiding this degeneration, DRNE takes node degree as the weakly guiding information, i.e., imposing a constraint that the learned embedding of a target node should be capable of approximating its degree. Consequently, the following regularizer is designed:

$$\mathcal{L}_{reg} = \sum_{v \in V} \|\log(d_v + 1) - MLP(Agg(\{\mathbf{X}_u | u \in \mathcal{N}(v)\}))\|_F^2 , \tag{20}$$

where the degree of node v is denoted by d_v and MLP stands for a single-layer multilayer perceptron taking the rectified linear unit (ReLU) [26] as activation function, defined as $ReLU(x) = max(0, x)$. DRNE minimizes the overall objective function by combining reconstruction loss in Eq. (12) and the regularizer in Eq. (20):

$$\mathcal{L} = \mathcal{L}_1 + \lambda \mathcal{L}_{reg}, \tag{21}$$

where λ is the balancing weight for regularizer. Note that degree information is not taken as the supervisory information for network embedding here. Alternatively, it is finally auxiliary to avoid degeneration. Therefore, the value of λ should be set small.

Neighborhood Sampling. The node degrees usually obey a heavy-tailed distribution [23] in real networks, i.e., the majority of nodes have very small degrees while a minor number of nodes have very high degrees. Inspired by this phenomenon, DRNE downsamples the neighbors of those large-degree nodes before feeding them into the ln-LSTM to improve the efficiency. In specific, an upper bound for the number of neighbors S is set. When the number of neighbors exceeds S, the neighbors are downsampled into S different nodes. An example on the sampling process is shown in Fig. 4a and b. In networks obeying power-law, more unique structural information are carried by the large-degree nodes than the common small-degree nodes. Therefore, a biased sampling strategy is adopted by DRNE to retain the large-degree nodes by setting $P(v) \propto d_v$, i.e., the probability of sampling neighbor node v being proportional to its degree.

3.2.3 Optimization

For purpose of optimizing DRNE, we need to minimize the total loss $\mathcal{L}$ as a function of the embeddings X and the neural network parameters set θ. These parameters are optimized by Adam [36]. The BackPropagation Through Time (BPTT) algorithm [71] estimates the derivatives. At the beginning of the training, the learning rate α for Adam is initialized to 0.0025. Algorithm 2 demonstrates the whole algorithm.

Algorithm 2 Deep Recursive Network Embedding

```
Require: the network $\mathbf{G} = (V, E)$
Ensure: the embeddings $\mathbf{X}$, updated neural network parameters set $\theta$
initial $\theta$ and $\mathbf{X}$ by random process
while the value of objective function do not converge do
    for a node $v \in E$ do
        downsampling $v$'s neighborhoods if its degree exceeds $S$
        sort the neighborhoods by their degrees
        fixed aggregator function, calculate partial derivative $\partial\mathcal{L}/\partial\mathbf{X}$ to update embeddings $\mathbf{X}$
        fixed embeddings, calculate partial derivative $\partial\mathcal{L}/\partial\theta$ to update $\theta$
    end for
end while
obtain the node representations $X$
```

3.2.4 Theoretical Analysis

It can be theoretically proved that the resulted embeddings from DRNE can reflect several common and typical centrality measures closely related to regular equivalence. In the following process of proof, the regularizer term in Eq. (20) for eliminating degeneration is ignored without loss of generality.

Theorem 1 *Eigenvector centrality [7], PageRank centrality [49] and degree centrality are three optimal solutions of DRNE, respectively.*

Lemma 1 *For any computable function, there exists a finite recurrent neural network (RNN) [45] that can compute it.*

Proof This is a direct consequence of Theorem 1 in [56]. □

Theorem 2 *If the centrality $C(v)$ of node v satisfies that $C(v) = \sum_{u\in\mathcal{N}(v)} F(u)C(u)$ and $F(v) = f(\{F(u), u \in \mathcal{N}(v)\})$ where f is any computable function, then $C(v)$ is one of the optimal solutions for DRNE.*

Proof For brevity, let us suppose that LSTM takes linear activation for all the activation function. This lemma is proved by showing that there exists a parameter setting $\{\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_o, \mathbf{W}_C, \mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o, \mathbf{b}_C\}$ in Eqs. (13), (14), (15) and (16) such that the node embedding $\mathbf{X}_u = [F(u), C(u)]$ is a fixed point. In fact, this parameter settings can be directly constructed. Suppose $\mathbf{W}_{a,i}$ denotes the i-th row of $\mathbf{W}_a$. With the input sequence $\{[F(u), C(u)], u \in \mathcal{N}(v)\}$, set $\mathbf{W}_{f,2}$ and $\mathbf{W}_{o,2}$ as $[0, 0]$, $\mathbf{W}_{i,2}$ as $[1, 0]$, $\mathbf{W}_{C,2}$ as $[0, 1]$, $\mathbf{b}_{f,2}$ and $\mathbf{b}_{o,2}$ as 1, $\mathbf{b}_{i,2}$ and $\mathbf{b}_{C,2}$as 0, then we can easily get $\mathbf{h}_{t,2} = \mathbf{o}_{f,2} * \mathbf{C}_{t,2} = \mathbf{C}_{t,2} = \mathbf{f}_{t,2} * \mathbf{C}_{t-1,2} + \mathbf{i}_{t,2} * \tilde{\mathbf{C}}_{t,2} = \mathbf{C}_{t-1,2} + F(t) * C(t)$. Hence, $\mathbf{h}_{T,2} = \sum_{u\in\mathcal{N}(v)} F(u)C(u) = C(v)$ where T is the length of the input sequence. Additionally, by Lemma 1, there exists a parameter setting $\{\mathbf{W}'_f, \mathbf{W}'_i, \mathbf{W}'_o, \mathbf{W}'_C, \mathbf{b}'_f, \mathbf{b}'_i, \mathbf{b}'_o, \mathbf{b}'_C\}$ to approximate f. By set $\mathbf{W}_{f,1}$ as $[\mathbf{W}'_f, 0]$, $\mathbf{W}_{o,1}$ as $[\mathbf{W}'_o, 0]$ and so on, we can get that $\mathbf{h}_{T,1} = f(\{F(u), u \in \mathcal{N}(v)\}) = F(v)$. Therefore $\mathbf{h}_T = [F(v), C(v)]$ and the node embedding $\mathbf{X}_v = [F(v), C(v)]$ is a fixed point. The proof is now completed. □

Table 2 Definition of centralities

Centrality	Definition $C(v)$	$F(v)$	$f(\{x_i\})$
Eigenvector	$1/\lambda * \sum_{u \in \mathcal{N}(v)} C(u)$	$1/\lambda$	Mean
PageRank	$\sum_{u \in \mathcal{N}(v)} 1/d_u * C(u)$	$1/d_v$	$1/(\sum I(x_i))$
Degree	$d_v = \sum_{u \in \mathcal{N}(v)} I(d_u)$	$1/d_v$	$1/(\sum I(x_i))$

We can easily conclude that eigenvector centrality, PageRank centrality, degree centrality satisfy the condition of Theorem 2 by the definitions of centralities in Table 2 with $(F(v), f(\{x_i\}))$, completing the proof for Theorem 1.

Based on Theorem 1, such a parameter setting of DRNE exists for any graph that the resulted embeddings are able to be one of the three centralities. This shows such expressive power of DRNE that different aspects of regular-equivalence-related network structural information are captured.

3.2.5 Analysis and Discussions

This section presents the out-of-sample extension and the complexity analysis.

Out-of-Sample Extension. For a node v newly arrived, we can feed the embeddings of its neighbors directly into the aggregating function to get the aggregated representation, i.e., the embedding of the new node through Eq. (12), if we know its connections to the existing nodes. Such a procedure has a complexity of $O(d_v k)$, where d_v stands for the degree of node v and k represents the length of embeddings.

Complexity Analysis. For a single node v in each iteration during the training procedure, the complexity of gradients calculation and parameters updating is $O(d_v k^2)$, where k stands for the length of embeddings abd d_v represents the degree of node v. The aforementioned sampling process keeps d_v from exceeding the bound S. Therefore, the total training complexity is $O(|V|Sk^2 I)$ where I stands for the number of iterations. k, the length of embeddings, usually takes a small number (e.g. 32, 64, 128). The upper bound S is 300 in DRNE. The number of iterations I normally takes a small number which is independent with $|V|$. Hence, the total time complexity of training procedure is actually linear to the number of nodes $|V|$.

4 Structure Preserving Hyper Network Embedding

Conventional pairwise networks are the scenarios of most network embedding methods, where each edge connects only a pair of nodes. Nonetheless, the relationships among data objects are much more complicated in real world applications, and they typically go beyond pairwise. For instance, Jack purchasing a coat with nylon material

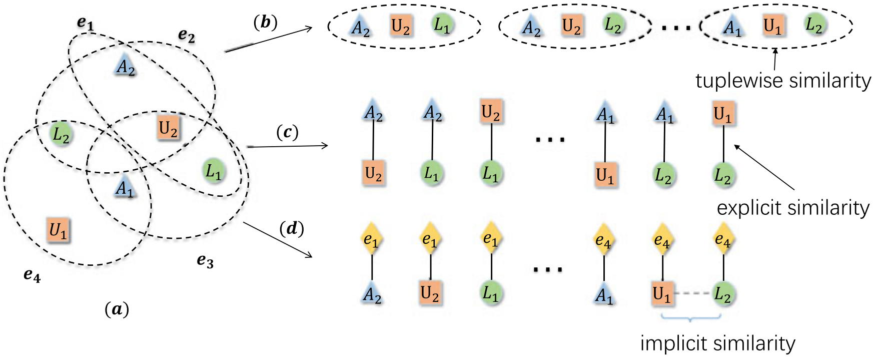


Fig. 5 **a** One example of a hyper-network. **b** DHNE. **c** The clique expansion. **d** The star expansion. DHNE models the hyperedge by and large, preserving the tuplewise similarity. In the case of clique expansion, each hyperedge is expanded into a clique, where each node pair is explicitly similar. In the case of the star expansion, each node of a hyperedge is linked to a new node representing the origin hyperedge, each node pair of which is implicitly similar in that they are linked to the same node. Figure from [65]

forms a high-order relationship ⟨Jack, coat, nylon⟩. A network designed to capture this kind of high-order node relationship is often known as a hyper-network.

For purpose of analyzing hyper-network, expanding them into traditional pairwise networks and then applying the analytical pairwise-network-based algorithms is a typical idea. Star expansion [1] (Fig. 5d) and clique expansion [60] (Fig. 5c) are two representative techniques of this category. For star expansion, a hypergraph is changed into a bipartite graph where each hyperedge is represented by an instance node linking to the original nodes contained by it. For clique expansion, a hyperedge is expanded as a clique. These methods make an assumption that the hyperedges are *decomposable* either implicitly or explicitly. In other words, if a set of nodes is treated as a hyperedge, then any subset of nodes contained by this hyperedge can constitute another hyperedge. This assumption is reasonable in homogeneous hyper-networks, because the constitution of hyperedges are caused by the latent similarity among the concerned objects, e.g. common labels, in most cases. Nonetheless, when it comes to the heterogeneous hyper-network embedding, it is essential to resolve the new demand as follows.

1. **Indecomposablity**: In heterogeneous hyper-networks, the hyperedges are often indecomposable. In the circumstances, a node set in a hyperedge has a strong inner relationship, whereas the nodes in its subset does not necessarily have. As an instance, in a recommendation system which has ⟨user, movie, tag⟩ relationships, the relationships of ⟨user, tag⟩ are often not strong. This phenomenon suggests that using those traditional expansion methods to simply decompose hyperedges does not make sense.
2. **Structure Preserving**: The observed relationships in network embedding can easily preserve local structures. Nevertheless, many existing relationships are

not observed owing to the sparsity of networks, when preserving hyper-network structures with only local structures is not sufficient. Some global structures like the neighborhood structures are employed to address this problem. Thus, how to simultaneously capture and preserve both global and local structures in hyper-networks still remains an open problem.

Tu et al. [65] put forward a deep hyper-network embedding (DHNE) model to deal with these challenges. To resolve the **Indecomposablity** issue, an indecomposable tuplewise similarity function is designed. The function is straightforward defined over the universal set of the nodes contained by a hyperedge, ensuring that the subsets of it are not contained in network embedding process. They provide theoretical proof that the indecomposable tuplewise similarity function cannot be linear. Consequently, they implement the tuplewise similarity function as a deep neural network with a non-linear activation function added, making it highly non-linear. To resolve the **Structure Preserving** issue, a deep autoencoder is designed to learn node representations by reconstructing neighborhood structure, making sure that the nodes which have alike neighborhood structures also have alike embeddings. To simultaneously address the two issues, the deep autoencoder are jointly optimized with the tuplewise similarity function.

4.1 Notations and Definitions

The key notations used by DHNE is illustrated in Table 3.

Definition 4.1 (*Hyper-network*) One **hyper-network** is a hypergraph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ where the set of nodes V belongs to T types $\mathbf{V} = \{\mathbf{V}_t\}_{t=1}^{T}$ and each hyperedge of the set of edges $\mathbf{E}$ may have more than two nodes $\mathbf{E} = \{E_i = (v_1, v_2, ..., v_{n_i})\}(n_i \geq 2)$. The hyper-network degenerates to a network when each hyperedge has only 2 nodes. The definition of the type of edge E_i is the combination of types of all the nodes in the edge. If $T \geq 2$, we define the hyper-network as **heterogeneous hyper-network**.

Table 3 Notations

Symbols	Meaning
T	Number of node types
$\mathbf{V} = \{\mathbf{V}_t\}_{t=1}^{T}$	Node set
$\mathbf{E} = \{(v_1, v_2, ..., v_{n_i})\}$	Hyperedge set
$\mathbf{A}$	Adjacency matrix of hyper-network
$\mathbf{X}_i^j$	Embedding of node i with type j
$\mathcal{S}(\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_N)$	N-tuplewise similarity function
$\mathbf{W}_j^{(i)}$	The i-th layer weight matrix with type j
$\mathbf{b}_j^{(i)}$	The i-th layer biases with type j

In order to obtain embedding in a hyper-network, it is necessary to preserve an indecomposable tuplewise relationship. The authors give a definition to the indecomposable structures as the first-order proximity of the hyper-network.

Definition 4.2 (*The First-order Proximity of Hyper-network*) **The first-order proximity** of a hyper-network measures the N-tuplewise similarity between nodes. If there exists a hyperedge among any N vertexes $v_1, v_2, ..., v_N$, the first-order proximity of these N vertexes is defined as 1. Note that this implies no first-order proximity for any subsets of these N vertexes.

In the real world, the first-order proximity suggests the indecomposable similarity among several entities. Moreover, real world networks are always sparse and incomplete, thus it is not sufficient to only consider first-order proximity for learning node embeddings. To address this issue, we need to consider higher order proximity. To capture the global structure, the authors then propose the definition of the second-order proximity of hyper-networks.

Definition 4.3 (*The Second-order Proximity of Hyper-network*) **The second-order Proximity** of a hyper-network measures the proximity of two nodes concerning their neighborhood structures. For any node $v_i \in E_i$, E_i/v_i is defined as a **neighborhood** of v_i. If v_i's neighborhoods $\{E_i/v_i \ for \ any \ v_i \in E_i\}$ are similar to v_j's, then v_i's embedding $\mathbf{x}_i$ should be similar to v_j's embedding $\mathbf{x}_j$.

For example, in Fig. 5a, A_1's neighborhood set is $\{(L_2, U_1), (L_1, U_2)\}$ and A_2's neighborhood set is $\{(L_2, U_2), (L_1, U_2)\}$. Thus A_1 and A_2 are second-order similar due to sharing common neighborhood (L_1, U_2).

4.2 *The DHNE Model*

This section presents the Deep Hyper-Network Embedding (DHNE) model, the framework of which is illustrated in Fig. 6.

4.2.1 Loss Function

For purpose of preserving the first-order proximity of hyper-networks, an N-tuplewise similarity measure is required in the embedding space. Such a measure should meet the requirement that when a hyperedge exists among N vertexes, the N-tuplewise similarity of them is supposed to be large and vice versa.

Property 1 *Let* $\mathbf{X}_i$ *denote the embedding of node* v_i *and* $\mathcal{S}$ *as* N*-tuplewise similarity function.*

- *if* $(v_1, v_2, ..., v_N) \in \mathbf{E}$, $\mathcal{S}(\mathbf{X}_1, \mathbf{X}_2, .., \mathbf{X}_N)$ *is supposed to be large (larger than a threshold l without loss of generality).*

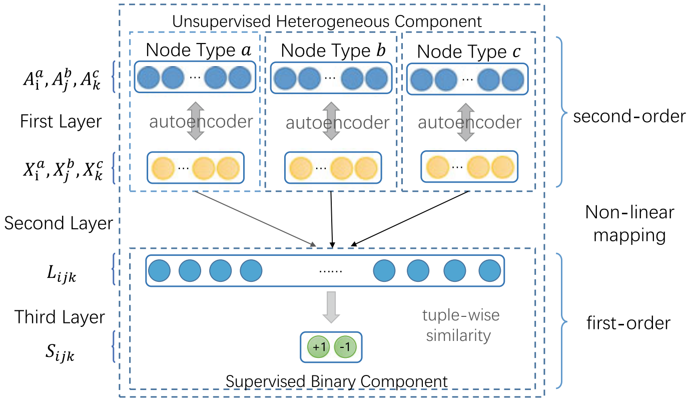


Fig. 6 Framework of Deep Hyper-Network Embedding (DHNE), figure from [65]

- *if* $(v_1, v_2, ..., v_N) \notin \mathbf{E}$, $\mathcal{S}(\mathbf{X}_1, \mathbf{X}_2, .., \mathbf{X}_N)$ *is supposed to be small (smaller than a threshold s without loss of generality).*

DHNE employs a data-dependent N-tuplewise similarity function and mainly focuses on hyperedges with uniform length $N = 3$, which is not difficult to extend to $N > 3$. The authors also propose a theorem to show that a linear tuplewise similarity function is not able to satisfy Property 1.

Theorem 3 *Linear function* $\mathcal{S}(\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_N) = \sum_i \mathbf{W}_i \mathbf{X}_i$ *cannot satisfy Property 1.*

Proof Using the counter-evidence method, let us presume that Theorem 3 is not true, i.e., the linear function $\mathcal{S}$ satisfies Property 1. Consider the counter example with 3 types of nodes where each type has 2 clusters (with the id 1 and 0). Hence, a hyperedge exists iff 3 nodes from different types have the same cluster id. We take the notation of $\mathbf{Y}_i^j$ to stand for embeddings of nodes with type j in cluster i. We hold the following inequations by Property 1:

$$\mathbf{W}_1\mathbf{Y}_0^1 + \mathbf{W}_2\mathbf{Y}_0^2 + \mathbf{W}_3\mathbf{Y}_0^3 > l \tag{22}$$

$$\mathbf{W}_1\mathbf{Y}_1^1 + \mathbf{W}_2\mathbf{Y}_0^2 + \mathbf{W}_3\mathbf{Y}_0^3 < s \tag{23}$$

$$\mathbf{W}_1\mathbf{Y}_1^1 + \mathbf{W}_2\mathbf{Y}_1^2 + \mathbf{W}_3\mathbf{Y}_1^3 > l \tag{24}$$

$$\mathbf{W}_1\mathbf{Y}_1^0 + \mathbf{W}_2\mathbf{Y}_1^2 + \mathbf{W}_3\mathbf{Y}_1^3 < s. \tag{25}$$

By combining Eqs. (22), (23), (24) and (25), we get $\mathbf{W}_1 * (\mathbf{Y}_0^1 - \mathbf{Y}_1^1) > l - s$ and $\mathbf{W}_1 * (\mathbf{Y}_1^1 - \mathbf{Y}_0^1) > l - s$. This is contradictory to our assumption and thus finishes the proof. □

Theorem 3 demonstrates that N-tuplewise similarity function $\mathcal{S}$ are supposed to be non-linear, which motivates DHNE to model the similarity by a multilayer perceptron. This contains two parts, illustrated separately in the 2nd layer and 3rd layer of Fig. 6, where the 2nd layer is a fully connected layer whose activation functions are non-linear. Inputted with the embeddings $(\mathbf{X}_i^a, \mathbf{X}_j^b, \mathbf{X}_k^c)$ of 3 nodes (v_i, v_j, v_k), they can be concatenated and mapped non-linearly to a common latent space $\mathbf{L}$ where the joint representation is shown as follows.

$$\mathbf{L}_{ijk} = \sigma(\mathbf{W}_a^{(2)} * \mathbf{X}_i^a + \mathbf{W}_b^{(2)} * \mathbf{X}_j^b + \mathbf{W}_c^{(2)} * \mathbf{X}_k^c + \mathbf{b}^{(2)}), \tag{26}$$

where σ stands for the sigmoid function. Finally, the latent representation $\mathbf{L}_{ijk}$ is mapped to a probability space in the 3rd layer to obtain the similarity:

$$\mathbf{S}_{ijk} \equiv \mathcal{S}(\mathbf{X}_i^a, \mathbf{X}_j^b, \mathbf{X}_k^c) = \sigma(\mathbf{W}^{(3)} * \mathbf{L}_{ijk} + \mathbf{b}^{(3)}). \tag{27}$$

Hence, the combination of the second and third layers can get a non-linear tuple-wise similarity measure function $\mathcal{S}$ as we hoped. For purpose of making $\mathcal{S}$ satisfy Property 1, we can write the following objective function.

$$\mathcal{L}_1 = -(\mathbf{R}_{ijk} \log \mathbf{S}_{ijk} + (1 - \mathbf{R}_{ijk}) \log(1 - \mathbf{S}_{ijk})), \tag{28}$$

where $\mathbf{R}_{ijk}$ is defined as 1 if there is a hyperedge between v_i, v_j and v_k and 0 otherwise. According to the objective function, it is not difficult to point out that if $\mathbf{R}_{ijk} = 1$, the similarity $\mathbf{S}_{ijk}$ is supposed to be large and vice versa. That is to say, the first-order proximity is successfully preserved.

The design of the first layer in Fig. 6 aims to preserve the second-order proximity, which measures the similarity of neighborhood structures. Here, to characterize the neighborhood structure, the authors define the adjacency matrix of hyper-network. Specifically, they define a $|\mathbf{V}| * |\mathbf{E}|$ incidence matrix $\mathbf{H}$ with elements $\mathbf{h}(v, e) = 1$ if $v \in e$ and 0 otherwise to denote a hypergraph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$. $d(v) = \sum_{e \in \mathbf{E}} \mathbf{h}(v, e)$ stands for the degree of a vertex $v \in \mathbf{V}$. Let $\mathbf{D}_v$ stand for the diagonal matrix containing the vertex degree, then the adjacency matrix $\mathbf{A}$ of hypergraph $\mathbf{G}$ can be defined as $\mathbf{A} = \mathbf{H}\mathbf{H}^T - \mathbf{D}_v$, where $\mathbf{H}^T$ is the transpose of $\mathbf{H}$. Here, each element of adjacency matrix $\mathbf{A}$ stands for the concurrent times between two nodes, while the i-th row of $\mathbf{A}$ demonstrates the neighborhood structure of vertex v_i. To make best of this information, DHNE utilizes an autoencoder [39] model to preserve the neighborhood structure and feeds it with an adjacency matrix $\mathbf{A}$ as the input feature. The autoencoder consists of two non-linear mapping: an encoder and a decoder, where the encoder maps from feature space $\mathbf{A}$ to latent representation space $\mathbf{X}$, while the decoder from latent representation $\mathbf{X}$ space back to origin feature space $\hat{\mathbf{A}}$, which can be shown as follows.

$$\mathbf{X}_i = \sigma(\mathbf{W}^{(1)} * \mathbf{A}_i + \mathbf{b}^{(1)}) \tag{29}$$

$$\hat{\mathbf{A}}_i = \sigma(\hat{\mathbf{W}}^{(1)} * \mathbf{X}_i + \hat{\mathbf{b}}^{(1)}). \tag{30}$$

The aim of autoencoder is to minimize the reconstruction error between the output and the input, with which process it will give similar latent representations to the nodes with similar neighborhoods, preserving the second-order proximity consequently. Note that the input feature, the adjacency matrix of the hyper-network, is often extremely sparse. To achieve a higher efficiency, DHNE only reconstructs non-zero elements in the adjacency matrix. The following equation shows the reconstruction error:

$$||sign(\mathbf{A}_i) \odot (\mathbf{A}_i - \hat{\mathbf{A}}_i)||_F^2, \tag{31}$$

where $sign$ stands for the sign function.

Additionally, in a heterogeneous hyper-network, the vertexes usually have various types, the distinct characteristics of which require the model to learn a unique latent space for each of them. Motivated by this idea, DHNE provides each heterogeneous type of entities with an autoencoder model of their own, as is demonstrated in Fig. 6. The definition of loss function for all types of nodes is as follows.

Algorithm 3 The Deep Hyper-Network Embedding (DHNE)

```
Require: the hyper-network $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, the adjacency matrix $\mathbf{A}$ and the parameter $\alpha$
Ensure: Hyper-network Embeddings $E$ and updated Parameters $\theta = \{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}, \hat{\mathbf{W}}^{(i)}, \hat{\mathbf{b}}^{(i)}\}_{i=1}^3$
Initialize parameters $\theta$ randomly
while the value of objective function has not converged do
    Generate the next batch from the set of hyperedges $\mathbf{E}$
    Sample negative hyperedge in a random way
    Compute partial derivative $\partial\mathcal{L}/\partial\theta$ with back-propagation to update $\theta$.
end while
```

$$\mathcal{L}_2 = \sum_t ||sign(\mathbf{A}_i^t) \odot (\mathbf{A}_i^t - \hat{\mathbf{A}}_i^t)||_F^2, \tag{32}$$

where t is the index for node types.

For purpose of simultaneously preserving first-order proximity and second-order proximity for a heterogeneous hyper-network, DHNE jointly minimizes the loss function via blending Eqs. (28) and (32):

$$\mathcal{L} = \mathcal{L}_1 + \alpha\mathcal{L}_2. \tag{33}$$

4.2.2 Optimization

DHNE adopts stochastic gradient descent (SGD) for optimization, the critical step of which is to compute the partial derivative of parameters $\theta = \{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}, \hat{\mathbf{W}}^{(i)}, \hat{\mathbf{b}}^{(i)}\}_{i=1}^3$. By back-propagation algorithm [39], these derivatives can be easily estimated. Note that in most real world networks, there exist only positive relationships, so that the iterative process may degenerate to trivial convergence where all the tuplewise relationships turn out to be similar. In order to resolve this issue, DHNE samples multiple negative edges with the help of a noisy distribution for each edge [46]. The whole algorithm is demonstrated in Algorithm 3.

4.2.3 Analysis and Discussions

This section presents the out-of-sample extension and the complexity analysis.

Out-of-sample extension For any new vertex v, it is easy to obtain the adjacency vector by this vertex's connections to other existing vertexes. Hence, the out-of-sample extension problem can be solved by feeding the new vertex v's adjacency vector into the specific autoencoder corresponded with its type and applying Eq. (29) to get its latent representation in embedding space. The time complexity for these steps is $\mathcal{O}(dd_v)$, where d stands for the dimensionality of the embedding space and d_v is the degree of vertex v.

Complexity analysis The complexity of gradients calculation and parameters updating in the training procedure is $O((nd + dl + l)bI)$, where n stands for the number of nodes, d represents the dimension of embedding vectors, l stands for the size of latent layer, b stands for the batch size and I represents the number of iterations. The parameter l is usually correlated with d, but independent on n and I also has no connection with n. b is normally small. Hence, the time complexity of the training procedure is actually linear to the number of vertexes n.

Deep Property-oriented Methods

5 Uncertainty-Aware Network Embedding

Usually, real-world networks, the constitution and evolution of which are full of uncertainties, can be much more sophisticated than we expect. There are many reasons resulting in such uncertainties. For instance, low-degree nodes in a network fail to provide enough information and hence the representations of them are more uncertain than others. For those nodes sharing numerous communities, the potential contradiction among its neighbors might also be larger than others, resulting in uncertainty. Moreover, in social networks, human behaviors are sophisticated, making the generation of edges also uncertain [72]. Therefore without considering the uncertainties in networks, the information of nodes may become incomplete in latent space, which makes the representations less effective for network analysis and inference. Nonetheless, previous work on network embedding mainly represents each node as a single point in lower-dimensional continuous vector space, which has a crucial limitation that it can not capture the uncertainty of the nodes. Given that the family of Gaussian methods are capable of innately modeling uncertainties [67] and provide various distance functions per object, it will be promising to represent a node with a Gaussian distribution so that the characteristics of uncertainties can be incorporated.

As such, to capture the uncertainty of each node during the process of network embedding with Gaussian process, there are several basic requirements. First, to preserve the transitivity in networks, the embedding space is supposed to be a metric space. Transitivity here is a significant property for networks, peculiarly social networks [32]. For instance, the possibility of a friend of my friend becoming my friend is much larger than that of a person randomly chosen from the crowd. Moreover, the transitivity measures the density of loops of length three (triangles) in networks, crucial for computing clustering coefficient and related attributes [12]. Importantly, the transitivity in networks can be preserved well on condition that the triangle inequality is satisfied by the metric space. Second, the uncertainties of nodes should be characterized by the variance terms so that these uncertainties can be well captured, which means that the variance terms should be explicitly related to mean vectors. In other words, the proximities of nodes are supposed to be captured by mean vectors, while the uncertainties of nodes are supposed to be modeled by variance terms. Third, network structures such as high-order proximity, which can be used in abundant real-world applications as shown in [48], are also supposed to be preserved effectively and efficiently.

Zhu et al. [73] propose a deep variational model, called **DVNE**, which satisfies the above requirements and learns the Gaussian embedding in the Wasserstein space. Wasserstein space [14] is a metric space where the learned representations are able to preserve the transitivity for networks well. Specifically, the similarity measure is defined as Wasserstein distance, a formidable tool based on the optimal transport theory for comparing data distributions with wide applications such as computer vision [8] and machine learning [37]. Moreover, the Wasserstein distance enables the fast computation for Gaussian distributions [25], taking linear time complexity to calculate the similarity between two node representations. Meanwhile, they use a variant of Wasserstein autoencoder (WAE) [64] to reflect the relationships between variance and mean terms, where WAE is a deep variational model which has the goal of minimizing the Wasserstein distance between the original data distribution and the predicted one. In general, via preserving the first-order and second-order proximity, the learned representations by DVNE is capable of well capturing the local and global network structures [61, 69].

5.1 Notations

$\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$ stands for a network, where $\mathbf{V} = \{v_1, v_2, ..., v_N\}$ is a set of nodes and N is the number of them. The set of edges between nodes is denoted as $\mathbf{E}$, where $M = |\mathbf{E}|$ is the number of them. Let $\mathbf{Nbrs}_i = \{v_j | (v_i, v_j) \in \mathbf{E}\}$ stand for the neighbors of v_i. The transition matrix is denoted as $\mathbf{P} \in \mathbb{R}^{N \times N}$, where $\mathbf{P}(:, j)$ and $\mathbf{P}(i, :)$ denote its *jth* column and *ith* row respectively. $\mathbf{P}(i, j)$ stands for the element at the *ith* row and *jth* column. Given that an edge links v_i to v_j and node degree of v_i is d_i, we set $\mathbf{P}(i, j)$ to $\frac{1}{d_i}$ and zero otherwise. Then, $\mathbf{h}_i = \mathcal{N}(\bar{}_i, \Sigma_i)$ is defined as a lower-dimensional

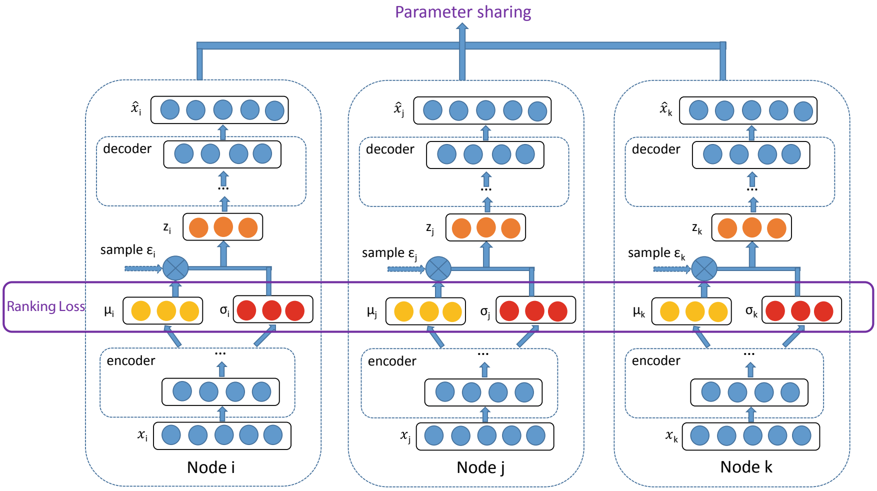


Fig. 7 The framework of DVNE, figure from [73]

Gaussian distribution embedding for node v_i, where $\mu_i \in \mathbb{R}^L$, $\Sigma_i \in \mathbb{R}^{L\times L}$. L stands for the embedding dimension, satisfying $L \ll N$.

5.2 *The DVNE Model*

The DVNE model proposed by Zhu et al. [73] is discussed in this section. The framework of DVNE is shown in Fig. 7.

5.2.1 Similarity Measure

For purpose of supporting network applications, a suitable similarity measure need to be defined between two node latent representations. In DVNE, distributions are adopted to model latent representations, thus the similarity measure here is supposed to be capable of measuring the similarities among different distributions. Moreover, the similarity measure is supposed to also simultaneously preserve the transitivity among nodes, since it is a crucial property of networks. Wasserstein distance is such an adequate candidate capable of measuring the similarity between two distributions and satisfying the triangle inequality simultaneously [13], which guarantees its capability of preserving the transitivity of similarities among nodes.

The definition of the *pth* Wasserstein distance between two probability measures ν and μ is

$$W_p(\nu, \mu)^p = \inf \mathbb{E}\big[d(X, Y)^p\big], \tag{34}$$

where $\mathbb{E}[Z]$ stands for the expectation of a random variable Z and inf stands for the infimum taken over all the joint distributions of the random variables X and Y, the marginals of which are ν and μ respectively.

Furthermore, it has been proved that the *pth* Wasserstein distance can preserve all properties of a metric when $p \geq 1$ [2]. The metric should satisfy the non-negativity, the symmetry, the identity of indiscernibles and the triangle inequality [10]. In such ways, Wasserstein distance meets the requirement of being a similarity measure for the latent node representations, peculiarly for an undirected network.

Although the computational cost of general-formed Wasserstein distance, which causes the limitation, a closed form solution can be achieved with the *2th* Wasserstein distance (abbreviated as W_2) since Gaussian distributions are used in DVNE for the latent node representations. This greatly reduces the computational cost.

More specifically, DVNE employs the following formula to calculate W_2 distance between two Gaussian distributions [25]:

$$\begin{aligned} dist &= W_2(\mathcal{N}(\mu_1, \Sigma_1), \mathcal{N}(\mu_2, \Sigma_2)) \\ dist^2 &= \|\mu_1 - \mu_2\|_2^2 + \mathrm{Tr}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1^{1/2}\Sigma_2\Sigma_1^{1/2})^{1/2}) \end{aligned} \tag{35}$$

Furthermore, the W_2 distance (a.k.a root mean square bipartite matching distance) has been popularly applied in computer graphics [9, 19], computer vision [8, 14] and machine learning [15, 16], etc. DVNE adopts diagonal covariance matrices,[4] thus $\Sigma_1\Sigma_2 = \Sigma_2\Sigma_1$. In the commutative case, the formula (35) can be simplified as

$$W_2\big(\mathcal{N}(\mu_1, \Sigma_1); \mathcal{N}(\mu_2, \Sigma_2)\big)^2 = \|\mu_1 - \mu_2\|_2^2 + \|\Sigma_1^{1/2} - \Sigma_2^{1/2}\|_F^2. \tag{36}$$

According to the above equation, the time complexity of computing W_2 distance between two nodes in latent embedding space is linear to L, the dimension of embedding.

5.2.2 Loss Functions

First, the first-order proximity needs to be preserved. Intuitively, each node connected with v_i is supposed to be of smaller distance to v_i in the embedding space compared to the nodes that have no edges linking v_i. More specifically, to preserve the first-order proximity, the following pairwise constraints is satisfied by DVNE:

$$W_2(\mathbf{h}_i, \mathbf{h}_j) < W_2(\mathbf{h}_i, \mathbf{h}_k), \forall v_i \in \mathbf{V}, \forall v_j \in \mathbf{Nbrs}_i, \forall v_k \notin \mathbf{Nbrs}_i. \tag{37}$$

[4] When the covariance matrices are not diagonal, Wang et al. propose a fast iterative algorithm (i.e., BADMM) to solve the Wasserstein distance [70].

The smaller the W_2 distance, the more similar between nodes. An energy based learning approach [40] is used here to satisfy all pairwise constraints which are defined above. The following equation presents the mathematical objective function, with $W_2(\mathbf{h}_i, \mathbf{h}_j)$ standing for the energy between two nodes, $E_{ij} = W_2(\mathbf{h}_i, \mathbf{h}_j)$.

$$\mathcal{L}_1 = \sum_{(i,j,k)\in\mathbf{D}} ({E_{ij}}^2 + exp(-E_{ik})), \tag{38}$$

where $\mathbf{D}$ stands for the set of all valid triplets given in Eq. (37). Therefore, ranking errors are penalized by the energy of the pairs in this objective function, making the energy of negative examples be higher than that of positive examples.

In order to preserve second-order proximity, transition matrix $\mathbf{P}$ is adopted as the input feature of Wasserstein Auto-Encoders (WAE) [64] to preserve the neighborhood structure and the mathematical relevance of mean vectors and variance terms is also implied. More specifically, $\mathbf{P}(i, :)$ demonstrates the neighborhood structure of node v_i, and is adopted as the input feature for node v_i to preserve its neighborhood structure. The objective of WAE contains the reconstruction loss and the regularizer, where the former loss helps to preserve neighborhood structure and latter guides the encoded training distribution to match the prior distribution. Let P_X denotes the data distribution, and P_G denotes the encoded training distribution, then the goal of WAE is minimizing Wasserstein distance between P_X and P_G. The reconstruction cost can be written as

$$D_{WAE}(P_X, P_G) = \inf_{Q(Z|X)\in Q} \mathbb{E}_{P_X}\mathbb{E}_{Q(Z|X)}\big[c(X, G(Z))\big], \tag{39}$$

where Q stands for the encoders and G represents the decoders, $X \sim P_X$ and $Z \sim Q(Z|X)$. According to [64], Eq. (39) minimizes the W_2 distance between P_X and P_G with $c(x, y) = \|x - y\|_2^2$. Taking the sparsity of transition matrix $\mathbf{P}$ into consideration, DVNE is centered on non-zero elements in $\mathbf{P}$ to accelerate the training process. Therefore, the loss function for preserving the second-order proximity can be defined as follows.

$$\mathcal{L}_2 = \inf_{Q(Z|X)\in Q} \mathbb{E}_{P_X}\mathbb{E}_{Q(Z|X)}\big[\|X\circ(X - G(Z))\|_2^2\big], \tag{40}$$

where $\circ$ denotes the element-wise multiplication. The transition matrix $\mathbf{P}$ is used as the input feature X in DVNE. The second-order proximity is then preserved by the reconstruction process through forcing nodes with similar neighborhoods to have similar latent representations.

For purpose of simultaneously preserving the first-order proximity and second-order proximity of networks, DVNE jointly minimizes the loss function of Eqs. (38) and (40) by combining them together:

$$\mathcal{L} = \mathcal{L}_1 + \alpha\mathcal{L}_2. \tag{41}$$

5.2.3 Optimization

Optimizing objective function (38) in large graphs is computationally expensive, which needs to compute all the valid triplets in $\mathbf{D}$. Hence, we uniformly sample triplets from $\mathbf{D}$, replacing $\sum_{(i,j,k)\in\mathbf{D}}$ with $\mathbb{E}_{(i,j,k)\sim\mathbf{D}}$ in Eq. (38). M triplets are sampled in each iteration from $\mathbf{D}$ to compute the estimation of gradient.

Z in objective function (40) is sampled from $Q(Z|X)$, which is a non-continuous operation without gradient. Similar to Variational Auto-Encoders (VAE) [21], the "reparameterization trick" is used here for the optimization of the above objective function via the deep neural networks. Firstly, we sample $\epsilon \sim \mathcal{N}(0, \mathbf{I})$. Then, we can calculate $Z = \mu(X) + \Sigma^{1/2}(X) * \epsilon$. Consequently, the objective function (40) becomes deterministic and continuous in the parameter spaces of encoders Q and decoders G, given a fixed X and ϵ, which means the gradients can be computed by backpropagation in deep neural networks.

5.2.4 Complexity Analysis

Algorithm 4 lists each step of DVNE. The complexity of gradient computation and parameters updating during the training procedure is $O(T * M * (d_{ave}S + SL + L))$, where T is the number of iterations, M stands for the number of edges, d_{ave} represents the average degree of all nodes, L stands for the dimension of embedding vectors and S represents the size of hidden layer. Because only non-zero elements in x_i are reconstructed in DVNE, the computational complexity of the first and last hidden layers is $O(d_{ave}S)$, while that of other hidden layers is $O(SL)$. In addition, computation of the W_2 distance among distributions takes $O(L)$. In experiments, convergence can be achieved with a small number of iterations T (e.g., $T \leq 50$).

Algorithm 4 DVNE Embedding

```
Require: The network $\mathbf{G} = \{\mathbf{V}, \mathbf{E}\}$ with the transition matrix $\mathbf{P}$, the parameter $\alpha$
Ensure: Network embeddings $\{\mathbf{h}_i\}_{i=1}^{N}$ and updated parameters $\theta = \{\mathbf{W}^{(i)}, \mathbf{b}^{(i)}\}_{i=1}^{5}$
Initialize parameters $\theta$ by xavier initialization
while $\mathcal{L}$ does not converge do
    Uniformly sample $M$ triplets from $\mathbf{D}$
    Split these triplets into a number of batches
    Compute partial derivative $\partial\mathcal{L}/\partial\theta$ with backpropagation algorithm to update $\theta$
end while
```

6 Dynamic-Aware Network Embedding

Despite the commendable success network embedding has achieved in tasks such as classification and recommendation, most existing algorithms in the literature to date are primarily designed for static networks, where all nodes are known before learning. However, for large-scale networks, it is infeasible to rerun network embedding whenever new nodes arrive, especially considering the fact that rerunning network embedding also results in the need of retraining all downstream classifiers. How to efficiently infer proper embeddings for out-of-sample nodes, i.e., nodes that arrive after the embedding process, remains largely unanswered.

Several graph-based methods in the literature can be adapted to infer out-of-sample embeddings given in-sample ones. Many of them deduce embeddings of new nodes by performing information propagation [74], or optimizing a loss that encourages smoothness between linked nodes [20, 75]. There are also methods that aim to learn a function mapping node features (e.g., text attributes, or rows of the adjacency matrix when attributes are unavailable) to outcomes/embeddings, while imposing a manifold regularizer derived from the graph [5]. The embeddings of out-of-sample nodes can then be predicted based on their features by these methods. Nevertheless, existing methods are facing several challenges. Firstly, the inferred embeddings of out-of-sample nodes should preserve intricate network properties with embeddings of in-sample nodes. For example, high-order proximity, among many other properties, is deemed especially essential to be preserved by network embedding [11, 48, 61], and thus must be reflected by the inferred embeddings. Secondly, as downstream applications (e.g., classification) will treat in-sample and out-of-sample nodes equally, the inferred embeddings and in-sample embeddings should possess similar characteristics (e.g., magnitude, mean, variance, etc.), i.e., belong to a homogeneous space, resulting in the need of a model expressive enough to characterize the embedding space. Finally, maintaining fast prediction speed is crucial, especially considering the highly dynamic nature of real-world networks. This final point is even more challenging due to the demand of simultaneously fulfilling the previous two requirements.

To infer out-of-sample embeddings, Ma et al. [44] propose a Deeply Transformed High-order Laplacian Gaussian Process (DepthLGP) approach (see Fig. 8) through

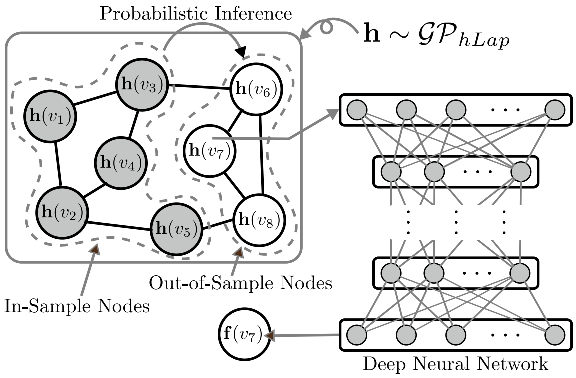


Fig. 8 Here v_6, v_7, and v_8 are out-of-sample nodes. Values of $\mathbf{h}(\cdot)$ are latent states. Values of shaded nodes are learned during training. To predict $\mathbf{f}(v_7)$, DepthLGP first predicts $\mathbf{h}(v_7)$ via probabilistic inference, then passes $\mathbf{h}(v_7)$ through a neural network to obtain $\mathbf{f}(v_7)$, figure from [44]

combining nonparametric probabilistic modeling with deep neural networks. More specifically, they first design a high-order Laplacian Gaussian process (hLGP) prior with a carefully constructed kernel that encodes important network properties such as high-order proximity. Each node is associated with a latent state that follows the hLGP prior. They then employ a deep neural network to learn a nonlinear transformation function from these latent states to node embeddings. The introduction of a deep neural network increases the expressive power of our model and improves the homogeneity of inferred embeddings with in-sample embeddings. Theories on the expressive power of DepthLGP are derived. Overall, their proposed DepthLGP model is fast and scalable, requiring zero knowledge of out-of-sample nodes during training process. The prediction routine revisits the evolved network rapidly and can produce inference results analytically with desirable time complexity linear with the number of in-sample nodes. DepthLGP is a general solution, in that it is applicable to embeddings learned by any network embedding algorithms.

6.1 The DepthLGP Model

In this section, we first formally formulate the out-of-sample node problem, and then discuss the DepthLGP model as well as theories on its expressive power.

6.1.1 Problem Definition

DepthLGP primarily considers undirected networks. Let $\mathcal{G}$ be the set of all possible networks and $\mathcal{V}$ be the set of all possible nodes. Given a specific network $G = (V, E) \in \mathcal{G}$ with nodes $V = \{v_1, v_2, \ldots, v_n\} \subset \mathcal{V}$ and edges E, a network embedding algorithm aims to learn values of a function $\mathbf{f} : \mathcal{V} \to \mathbb{R}^d$ for nodes in V. As the network evolves over time, a batch of m new nodes $V^* = \{v_{n+1}, v_{n+2}, \ldots, v_{n+m}\} \subset \mathcal{V} \setminus V$ arrives, and expands G into a larger network $G' = (V', E')$, where $V' = V \cup V^*$. Nodes in V^* are called out-of-sample nodes. The problem, then, is to infer values of $\mathbf{f}(v)$ for $v \in V^*$, given $G' = (V', E')$ and $\mathbf{f}(v)$ for $v \in V$.

6.1.2 Model Description

DepthLGP first assumes that there exists a latent function $\mathbf{h} : \mathcal{V} \to \mathbb{R}^s$, and the embedding function $\mathbf{f} : \mathcal{V} \to \mathbb{R}^d$ is transformed from the said latent function. To be more specific, let $\mathbf{g} : \mathbb{R}^s \to \mathbb{R}^d$ be the transformation function. DepthLGP then assumes that $\mathbf{f}(v) = \mathbf{g}(\mathbf{h}(v))$ for all $v \in \mathcal{V}$. Since the transformation can potentially be highly nonlinear, the authors use a deep neural network to serve as $\mathbf{g}(\cdot)$. DepthLGP further assumes that the s output dimensions of $\mathbf{h}(\cdot)$, i.e. $h_k : \mathcal{V} \to \mathbb{R}$

for $k = 1, 2, \ldots, s$, can be modeled independently. In other words, DepthLGP deals with each $h_k(v)$ of $\mathbf{h}(v) = \big[h_1(v), h_2(v), \ldots, h_s(v)\big]^\top$ separately.

Let us focus on $h_k(\cdot)$ for the moment. Each $h_k(\cdot)$ is associated with a kernel that measures similarity between nodes of a network. Take $G' = (V', E')$ with $V' = \{v_1, v_2, \ldots, v_{n+m}\}$ for example, the said kernel produces a kernel matrix $\mathbf{K}_k \in \mathbb{R}^{(n+m)\times(n+m)}$ for G':

$$
\begin{aligned}
\mathbf{K}_k &\triangleq \Big[\mathbf{I} + \eta_k \mathbf{L}(\hat{\mathbf{A}}_k) + \zeta_k \mathbf{L}(\hat{\mathbf{A}}_k\hat{\mathbf{A}}_k)\Big]^{-1}, \\
\hat{\mathbf{A}}_k &\triangleq \mathrm{diag}(\boldsymbol{\alpha}_k)\mathbf{A}'\mathrm{diag}(\boldsymbol{\alpha}_k), \\
\boldsymbol{\alpha}_k &\triangleq [a^{(k)}_{v_1}, a^{(k)}_{v_2}, \ldots, a^{(k)}_{v_{n+m}}]^\top,
\end{aligned}
$$

where $\mathbf{A}'$ is the adjacency matrix of G', while $\eta_k \in [0, \infty)$, $\zeta_k \in [0, \infty)$ and $a^{(k)}_v \in [0, 1]$ for $v \in \mathcal{V}$ are parameters of the kernel. $\mathrm{diag}(\cdot)$ returns a diagonal matrix corresponding to its vector input, while $\mathbf{L}(\cdot)$ treats its input as an adjacency matrix and returns the corresponding Laplacian matrix, i.e., $\mathbf{L}(\mathbf{A}) = \mathrm{diag}(\sum_i \mathbf{A}_{:,i}) - \mathbf{A}$.

The parameters of the proposed kernel have clear physical meanings. η_k indicates the strength of first-order proximity (i.e., connected nodes are likely to be similar), while ζ_k is for second-order proximity (i.e., nodes with common neighbors are likely to be similar). On the other hand, $a^{(k)}_v$ represents a node weight, i.e., how much attention we should pay to node v when conducting prediction. Values of $a^{(k)}_v$ for in-sample nodes ($v \in V$) are learned along with η_k and ζ_k (as well as parameters of the neural network $\mathbf{g}(\cdot)$) during training, while values of $a^{(k)}_v$ for out-of-sample nodes ($v \in V^*$) are set to 1 during prediction, since we are always interested in these new nodes when inferring embeddings for them. Node weights help DepthLGP avoid uninformative nodes. For example, in a social network, this design alleviates harmful effects of “bot” users that follow a large amount of random people and spam uninformative contents.

It is easy to see that $\mathbf{K}_k$ is positive definite, hence a valid kernel matrix. The kernel in DepthLGP can be seen as a generalization of the regularized Laplacian kernel [57], in that the authors further introduce node weighting and a second-order term. This kernel is referred as the high-order Laplacian kernel.

DepthLGP assumes that each sub-function $h_k(\cdot)$ follows a zero mean Gaussian process (GP) [51] parameterized by the high-order Laplacian kernel, i.e., $h_k \sim \mathcal{GP}^{(k)}_{hLap}$. This is equivalent to say that: For any $G_t = (V_t, E_t) \in \mathcal{G}$ with $V_t = \{v^{(t)}_1, v^{(t)}_2, \ldots, v^{(t)}_{n_t}\} \subset \mathcal{V}$, we have

$$
[h_k(v^{(t)}_1), h_k(v^{(t)}_2), \ldots, h_k(v^{(t)}_{n_t})]^\top \sim \mathcal{N}(\mathbf{0}, \mathbf{K}^{(t)}_k),
$$

where $\mathbf{K}^{(t)}_k$ is the corresponding high-order Laplacian kernel matrix computed on G_t.

The DepthLGP model can be summarized as follows.

$$\begin{aligned} h_k &\sim \mathcal{GP}^{(k)}_{hLap}, & k = 1, 2, \ldots, s, \\ \mathbf{h}(v) &\triangleq [h_1(v), h_2(v), \ldots, h_s(v)]^\top, & v \in \mathcal{V}, \\ \mathbf{f}(v) \mid \mathbf{h}(v) &\sim \mathcal{N}(\mathbf{g}(\mathbf{h}(v)), \sigma^2\mathbf{I}), & v \in \mathcal{V}. \end{aligned}$$

where σ is a hyper-parameter to be manually specified. The neural network $\mathbf{g}(\cdot)$ is necessary here, since $\mathbf{f}(\cdot)$ itself might not follow the GP prior exactly. The introduction of $\mathbf{g}(\cdot)$ allows the model to fit $\mathbf{f}(\cdot)$ more accurately.

6.1.3 Prediction

Before new nodes arrive, we have the initial network $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$, and know the values of $\mathbf{f}(v)$ for $v \in V$. The prediction routine assumes that there is a training procedure (see Sect. 6.1.4) conducted on G and $\mathbf{f}(v)$ for $v \in V$ before new nodes arrive, and the training procedure learns η_k, ζ_k, $h_k(v)$, $a^{(k)}_v$ for $k = 1, 2, \ldots, s$ and $v \in V$, as well as parameters of the transformation function $\mathbf{g}(\cdot)$.

As the network evolves over time, m new nodes $V^* = \{v_{n+1}, v_{n+2}, \ldots, v_{n+m}\}$ arrive and G evolves into $G' = (V', E')$ with $V' = V \cup V^*$. DepthLGP's prediction routine aims to predict $\mathbf{f}(v)$ for $v \in V^*$ by maximizing $p(\{\mathbf{f}(v) : v \in V^*\} \mid \{\mathbf{f}(v) : v \in V\}, \{\mathbf{h}(v) : v \in V\})$, which, according to our model, is equal to

$$p(\{\mathbf{f}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V\}).$$

Yet, it requires integrating over all possible $\mathbf{h}(v)$ for $v \in V^*$. DepthLGP therefore approximates it by maximizing

$$p(\{\mathbf{f}(v) : v \in V^*\}, \{\mathbf{h}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V\}),$$

which is equal to

$$\begin{aligned} &p(\{\mathbf{f}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V^*\}) \\ &\quad \times p(\{\mathbf{h}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V\}). \end{aligned}$$

It can be maximized[5] by first maximizing the second term, i.e. $p(\mathbf{h}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V\}) = \prod_{k=1}^{s} p(\{h_k(v) : v \in V^*\} \mid \{h_k(v) : v \in V\})$, and then setting $\mathbf{f}(v) = \mathbf{g}(\mathbf{h}(v))$ for $v \in V^*$.

[5] Note that the first term $p(\{\mathbf{f}(v) : v \in V^*\} \mid \{\mathbf{h}(v) : v \in V^*\})$ is maximized with $\mathbf{f}(v) = \mathbf{g}(\mathbf{h}(v))$, and the maximum value of this probability density is a *constant* unrelated with $\mathbf{h}(v)$. Hence we can focus on maximizing the second term first.

Algorithm 5 DepthLGP's Prediction Routine

```
Require: $G' = (V', E')$    ▷ $G = (V, E)$ evolves into $G'$.
Ensure: predicted values of $\mathbf{f}(v)$, $v \in V^*$    ▷ $V^* \triangleq V' \setminus V$.
▷ Let $V = \{v_1, v_2, \dots, v_n\}$ be old nodes.
▷ Let $V^* = \{v_{n+1}, v_{n+2}, \dots, v_{n+m}\}$ be new nodes.
▷ Let $\mathbf{A}'$ be the adjacency matrix of $G'$.
for $k = 1, 2, \dots, s$ do
    ▷ Values of $a_v^{(k)}$ are set to 1 for $v \in V^*$.
    $\boldsymbol{\alpha} \leftarrow [a_{v_1}^{(k)}, a_{v_2}^{(k)}, \dots, a_{v_{n+m}}^{(k)}]^\top$
    $\hat{\mathbf{A}} \leftarrow \mathrm{diag}(\boldsymbol{\alpha})\ \mathbf{A}\ \mathrm{diag}(\boldsymbol{\alpha})$
    ▷ Function $\mathbf{L}(\cdot)$ below treats $\hat{\mathbf{A}}$ and $\hat{\mathbf{A}}\hat{\mathbf{A}}$ as adjacency matrices, and returns their Laplacian matrices.
    $\mathbf{M} \leftarrow \mathbf{I} + \eta_k \mathbf{L}(\hat{\mathbf{A}}) + \zeta_k \mathbf{L}(\hat{\mathbf{A}}\hat{\mathbf{A}})$
    $\mathbf{M}_{*,*} \leftarrow$ the bottom-right $m \times m$ block of $\mathbf{M}$
    $\mathbf{M}_{*,x} \leftarrow$ the bottom-left $m \times n$ block of $\mathbf{M}$
    ▷ Let $\mathbf{z}_x^{(k)} \triangleq [h_k(v_1), h_k(v_2), \dots, h_k(v_n)]^\top$.
    ▷ Compute $\mathbf{M}_{*,x}\mathbf{z}_x^{(k)}$ first below for efficiency.
    $\mathbf{z}_*^{(k)} \leftarrow -\mathbf{M}_{*,*}^{-1}\mathbf{M}_{*,x}\mathbf{z}_x^{(k)}$    ▷ $\mathbf{z}_*^{(k)}$ is a prediction of $[h_k(v_{n+1}), h_k(v_{n+2}), \dots, h_k(v_{n+m})]^\top$.
end for
for $v \in V^*$ do
    ▷ Previous lines have produced a prediction of $\mathbf{h}(v) = [h_1(v), h_2(v), \dots, h_s(v)]^\top$. The line below now uses the said prediction to further predict $\mathbf{f}(v)$.
    compute $\mathbf{g}(\mathbf{h}(v))$    ▷ It is a prediction of $\mathbf{f}(v)$.
end for
```

Let us now focus on the subproblem, i.e., maximizing

$$p(\{h_k(v) : v \in V^*\} \mid \{h_k(v) : v \in V\}). \tag{42}$$

Since $h_k \sim \mathcal{GP}_{hLap}$, by definition we have

$$\begin{aligned}[h_k(v_1), h_k(v_2), \dots, &h_k(v_n), h_k(v_{n+1}), \dots, h_k(v_{n+m})]^\top \\ &\sim \mathcal{N}(\mathbf{0}, \mathbf{K}_k),\end{aligned}$$

where $\mathbf{K}_k$ is the corresponding kernel matrix computed on G'. We then have the following result:

$$\begin{aligned}\mathbf{z}_*^{(k)} \mid \mathbf{z}_x^{(k)} &\sim \mathcal{N}(\mathbf{K}_{*,x}\mathbf{K}_{x,x}^{-1}\mathbf{z}_x^{(k)}, \mathbf{K}_{*,*} - \mathbf{K}_{*,x}\mathbf{K}_{x,x}^{-1}\mathbf{K}_{*,x}^\top), \\ \mathbf{z}_x^{(k)} &\triangleq \left[h_k(v_1), h_k(v_2), \dots, h_k(v_n)\right]^\top, \\ \mathbf{z}_*^{(k)} &\triangleq \left[h_k(v_{n+1}), h_k(v_{n+2}), \dots, h_k(v_{n+m})\right]^\top,\end{aligned}$$

where $\mathbf{K}_{x,x}$, $\mathbf{K}_{*,x}$, and $\mathbf{K}_{*,*}$ are respectively the top-left $n \times n$, bottom-left $m \times n$, and bottom-right $m \times m$ blocks of $\mathbf{K}_k$. Though $\mathbf{K}_{*,x}\mathbf{K}_{x,x}^{-1}\mathbf{z}_x^{(k)}$ is expensive to compute, it can thankfully be proved to be equivalent to $-\mathbf{M}_{*,*}^{-1}\mathbf{M}_{*,x}\mathbf{z}_x^{(k)}$, where $\mathbf{M}_{*,x}$ and $\mathbf{M}_{*,*}$ are respectively the bottom-left $m \times n$ and bottom-right $m \times m$ blocks of $\mathbf{K}_k^{-1}$. $\mathbf{K}_k^{-1}$

is cheap to obtain as the matrix inversion gets cancelled out. And computing $\mathbf{M}_{*,*}^{-1}$ is fast, since $m \ll n$. As a result, Eq. (42) is maximized as:

$$\mathbf{z}_{*}^{(k)} = -\mathbf{M}_{*,*}^{-1}\mathbf{M}_{*,x}\mathbf{z}_{x}^{(k)}.$$

As a side node, maximizing Eq. (42) is in fact equivalent to minimizing the following criterion:

$$\begin{aligned} &\sum_{u \in V'} [h_k(u)]^2 + \\ &\frac{1}{2}\eta_k \sum_{u,v \in V'} a_u^{(k)} A'_{uv} a_v^{(k)} \left[h_k(u) - h_k(v)\right]^2 + \\ &\frac{1}{2}\zeta_k \sum_{u,v,w \in V'} a_u^{(k)} A'_{uw} a_w^{(k)} a_w^{(k)} A'_{wv} a_v^{(k)} \left[h_k(u) - h_k(v)\right]^2 , \end{aligned}$$

where A'_{uv} is the edge weight (zero if not connected) between u and v in G'. This form hints at the physical meanings of η, ζ and $a_v^{(k)}$ from another perspective.

The prediction routine is summarized in Algorithm 5.

6.1.4 Training

Training is conducted on the initial network, i.e., $G = (V, E)$, with the values of $\mathbf{f}(v)$ for $v \in V$. Since it does not depend on the evolved network $G' = (V', E')$, it can be carried out before new nodes arrive. It aims to find suitable parameters of the neural network $\mathbf{g}(\cdot)$ and proper values of η_k, ζ_k, $a_v^{(k)}$, $h_k(v)$ for $v \in V$ and $k = 1, 2, \ldots, s$.

The authors apply empirical minimum risk (ERM) training to DepthLGP model. ERM training of a probabilistic model, though not as conventional as maximum likelihood estimation (MLE) and maximum a posteriori (MAP) estimation, has been explored by many researchers before, e.g., [59]. Using ERM training here eliminates the need to specify σ, and is faster and more scalable as it avoids computing determinants.

The training procedure is listed in Algorithm 6. The basic idea is to first sample some subgraphs from G, then treat a small portion of nodes in each subgraph as if they were out-of-sample nodes, and minimize empirical risk on these training samples (i.e., minimize mean squared error of predicted embeddings).

Now let us describe how each training sample, $G'_t = (V'_t, E'_t)$, is sampled. DepthLGP first samples a subset of nodes, V_t^*, from G, along a random walk path. Nodes in V_t^* are treated as "new" nodes. DepthLGP then samples a set of nodes, V_t, from the neighborhood of V_t^*. DepthLGP defines the neighborhood of V_t^* to be nodes that are no more than two steps away from V_t^*. Finally, let $V'_t = V_t^* \cup V_t$, and G'_t be the subgraph induced in G by V'_t.

Optimization is be done with a gradient-based method and the authors use Adam [36] for this purpose. Gradients are computed using back-propagation [22, 53]. Good parameter initialization can substantially improve convergence speed. To allow easy initialization, they use a residual network (more strictly speaking, a residual block) [28] to serve as $\mathbf{g}(\cdot)$. In other words, DepthLGP chooses $\mathbf{g}(\cdot)$ to be of the form $\mathbf{g}(\mathbf{x}) = \mathbf{x} + \tilde{\mathbf{g}}(\mathbf{x})$, where $\tilde{\mathbf{g}}(\cdot)$ is a feed-forward neural network. In this case, $s = d$. Thus it is able to initialize values of $\mathbf{h}(v)$ to be values of $\mathbf{f}(v)$ for nodes in V.

6.1.5 On the Expressive Power of DepthLGP

Theorem 4 below demonstrates the expressive power of DepthLGP, i.e., to what degree it can model arbitrary $\mathbf{f} : \mathcal{V} \to \mathbb{R}^d$.

Theorem 4 (Expressive Power) *For any $\epsilon > 0$, any nontrivial $G = (V, E)$ and any $\mathbf{f} : \mathcal{V} \to \mathbb{R}^d$, there exists a parameter setting for DepthLGP, such that: For any $v^* \in V$, after deleting all information (except G) related with v^*, DepthLGP can still recover $\mathbf{f}(v^*)$ with error less than ϵ, by treating v^* as a new node and using Algorithm 5 on G.*

Remark A nontrivial G means that all connected components of G have at least three nodes. Information related with v^* includes $\mathbf{f}(v^*)$, $h_k(v^*)$ and $a_{v^*}^{(k)}$ for $k = 1, 2, \ldots, s$ (note that during prediction, $a_{v^*}^{(k)}$ is replaced by 1 since v^* is treated as a new node). Error is expressed in terms of l^2-norm. It can be proved with a constructive proof based on the universal approximation property of neural networks [17, 33].

Algorithm 6 DepthLGP's Training Routine

```
Require: $G = (V, E)$;  $\mathbf{f}(v)$ for $v \in V$
Ensure: $\eta_k, \zeta_k, a_v^{(k)}, h_k(v)$ for $v \in V$ and $k = 1, 2, \ldots, s$;  parameters of the neural network $\mathbf{g}(\cdot)$
for $t = 1, 2, \ldots, T$ do
    ▷ See Sect. 6.1.4 for more details on how to sample $V_t^*$ and $V_t$.
    $V_t^* \leftarrow$ a few nodes sampled along a random walk
    $V_t \leftarrow$ some nodes in $V_t^*$'s neighborhood
    $V_t' \leftarrow V_t \cup V_t^*$
    $G_t' \leftarrow$ the subgraph induced in $G$ by $V_t'$
    Execute Algorithm 5, but using $G_t'$ in place of $G'$, $V_t$ in place of old nodes, and $V_t^*$ in place
  of new nodes. Save its prediction of $\mathbf{f}(v)$ as $\tilde{\mathbf{f}}(v)$ for $v \in V_t^*$.
    loss $\leftarrow \frac{1}{|V_t^*|} \sum_{v \in V_t^*} \|\mathbf{f}(v) - \tilde{\mathbf{f}}(v)\|_{l^2}^2$
    Use back-propagation to compute the gradient of the loss with respect to $\eta_k, \zeta_k, a_v^{(k)}, h_k(v)$
  for $v \in V_t$ and parameters of $\mathbf{g}(\cdot)$.
    Apply gradient descent.
end for
```

Theorem 5 below then emphasizes the importance of second-order proximity: Even though DepthLGP leverages the expressive power of a neural network, modeling second-order proximity is still necessary.

Theorem 5 (On Second-Order Proximity) *Theorem 4 will not hold if DepthLGP does not model second-order proximity. That is, there will exist $G = (V, E)$ and $\mathbf{f} : \mathcal{V} \to \mathbb{R}^d$ that DepthLGP cannot model, if ζ_k is fixed to zero.*

6.2 Extensions and Variants

6.2.1 Integrating into an Embedding Algorithm

DepthLGP can also be easily incorporated into an existing network embedding algorithm to derive a new embedding algorithm capable of addressing out-of-sample nodes. Take node2vec [27] for example. For an input network $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, node2vec's training objective can be abstracted as

$$\min_{\theta, \mathbf{F}} \mathcal{L}_\theta(\mathbf{F}, G),$$

where columns of $\mathbf{F} \in \mathbb{R}^{d \times n}$ are target node embeddings to be learned, and θ contains parameters other than $\mathbf{F}$.

Let us use $\mathbf{f}_\phi : \mathcal{V} \to \mathbb{R}^d$ to represent a function parameterized by ϕ. This function is defined as follows. For $v \in V$, it first samples nodes from v's neighborhood (see Sect. 6.1.4 on how to sample them) and induces a subgraph from G containing v and these sampled nodes. It then treats v as a new node, nodes sampled from v's neighborhood as old nodes, and runs Algorithm 5 on the induced subgraph to obtain a prediction of v's embedding—this is the value of $\mathbf{f}_\phi(v)$. By definition, ϕ contains parameters of a neural network, η_k, ζ_k, $a_v^{(k)}$, and $h_k(v)$ for $v \in V$, $k = 1, 2, \ldots, s$. To derive a new embedding algorithm based on node2vec, we can simply change the training objective to:

$$\min_{\theta, \phi} \mathcal{L}_\theta([\mathbf{f}_\phi(v_1), \mathbf{f}_\phi(v_2), \ldots, \mathbf{f}_\phi(v_n)], G).$$

The authors name this new algorithm node2vec++, where $\mathbf{f}_\phi(v)$ is node v's embedding. Clearly, node2vec++ can handle out-of-sample nodes efficiently in the same fashion as DepthLGP.

6.2.2 Efficient Variants

When predicting node embeddings for out-of-sample nodes, DepthLGP can collectively infer all of them in one pass. However, if the number of newly arrived nodes is large, it is more efficient (and more memory-saving) to process new nodes in a batch-by-batch way: For each unprocessed new node v, find the largest connected component containing v and other new nodes (but not old nodes). Let V_t^* be nodes in the connected component, and V_t be old nodes sampled from V_t^*'s neighborhood (see Sect. 6.1.4 on how to sample them). Then it is possible to run Algorithm 5 on the subgraph induced by $V_t^* \cup V_t$ to obtain prediction for new nodes in V_t^*. Repeat this process until all new nodes are processed.

Some simplifications can be made to DepthLGP without sacrificing much performance while allowing faster convergence and a more efficient implementation of Algorithm 5. In particular, sharing node weights across different dimensions, i.e. keeping $a_v^{(1)} = \ldots = a_v^{(s)}$, hurts little for most mainstream embedding algorithms (though theoretically it will reduce the expressive power of DepthLGP). Similarly, for node2vec and DeepWalk, we can keep $\eta_1 = \ldots = \eta_s$ and $\zeta_1 = \ldots = \zeta_s$, since they treat different dimensions of node embeddings equally. For LINE, however, it is better to keep $\eta_1 = \ldots = \eta_{\frac{s}{2}}$ ($\zeta_1 = \ldots = \zeta_{\frac{s}{2}}$) and $\eta_{\frac{s}{2}+1} = \ldots = \eta_s$ ($\zeta_{\frac{s}{2}+1} = \ldots = \zeta_s$) separately, because an embedding produced by LINE is the result of concatenating two sub-embeddings (for 1st- and 2nd-order proximity respectively).

7 Conclusion and Future Work

This chapter introduces the problem of graph representation/network embedding and the challenges lying in the literature, i.e., high non-linearity, structure-preserving, property-preserving and sparsity. Given its success in handling large-scale non-linear data in the past decade, we remark that deep neural network (i.e., deep learning) serves as an adequate candidate to tackle these challenges and highlight the promising potential for combining graph representation/network embedding with deep neural network. We select five representative models on deep network embedding/graph representation for discussions, i.e., structural deep network embedding (SDNE), deep variational network embedding (DVNE), deep recursive network embedding (DRNE), deeply transformed high-order Laplacian Gaussian process (DepthLGP) based network embedding and deep hyper-network embedding (DHNE). In particular, SDNE and DRNE focus on structure-aware network embedding, which preserve the high order proximity and global structure respectively. By extending vertex pairs to vertex tuples, DHNE targets at learning embeddings for vertexes with various types in heterogeneous hyper-graphs and preserving the corresponding hyper structures. DVNE focuses on the uncertainties in graph representations and DepthLGP aims to learn accurate embeddings for new nodes in dynamic networks. Our discussions center around two aspects in graph representation/network embedding (i) deep structure-oriented network embedding and (ii) deep property-oriented network embedding. We hope that readers may benefit from our discussions.

The above discussions of the state-of-the-art network embedding algorithms highly demonstrate that the research field of network embedding is still young and promising. Selecting appropriate methods is a crucial question for tackling practical applications with network embedding. The foundation here is the property and structure preserving issue. Serious information in the embedding space may lost if the important network properties cannot be retained and the network structure cannot be well preserved, which damages the analysis in the sequel. The off-the-shelf machine learning methods can be applied based on the property and structure preserving network embedding. Available side information can be fed into network embedding. Moreover, for some certain applications, the domain knowledge can also be introduced as advanced information. At last, existing network embedding methods are mostly designed for static networks, while not surprisingly, many networks in real world applications are evolving over time. Therefore, novel network embedding methods to deal with the dynamic nature of evolving networks are highly desirable.

Acknowledgements We thank Ke Tu (DRNE and DHNE), Daixin Wang (SDNE), Dingyuan Zhu (DVNE) and Jianxin Ma (DepthLGP) for providing us with valuable materials.
Xin Wang is the corresponding author. This work is supported by China Postdoctoral Science Foundation No. BX201700136, National Natural Science Foundation of China Major Project No. U1611461 and National Program on Key Basic Research Project No. 2015CB352300.

References

1. Agarwal, S., Branson, K., Belongie S.: Higher order learning with graphs. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 17–24. ACM (2006)
2. Ambrosio, L., Gigli, N., Savaré, G.: Gradient Flows: in Metric Spaces and in the Space of Probability Measures. Springer Science & Business Media (2008)
3. Ba, J.L. Kiros, J.R., Hinton, G.E.: Layer normalization (2016). arXiv preprint arXiv:1607.06450
4. Belkin, M., Niyogi, P.: Laplacian eigenmaps for dimensionality reduction and data representation. Neural Comput. **15**(6), 1373–1396 (2003)
5. Belkin, M., Niyogi, P., Sindhwani, V.: Manifold regularization: a geometric framework for learning from labeled and unlabeled examples. J. Mach. Learn. Res. **7**, 2399–2434 (2006)
6. Bengio, Y.: Learning deep architectures for AI. Found. Trends® Mach. Learn. **2**(1), 1–127 (2009)
7. Bonacich, P.: Some unique properties of eigenvector centrality. Soc. Netw. **29**(4), 555–564 (2007)
8. Bonneel, N., Rabin, J., Peyré, G., Pfister, H.: Sliced and radon wasserstein barycenters of measures. J. Math. Imaging Vis. **51**(1), 22–45 (2015)
9. Bonneel, N., Van De Panne, M., Paris, S., Heidrich, W.: Displacement interpolation using lagrangian mass transport. ACM Trans. Graph. (TOG) **30**, 158. ACM (2011)
10. Bryant, V.: Metric Spaces: Iteration and Application. Cambridge University Press (1985)
11. Cao, S., Lu, W., Xu, Q.: Grarep: learning graph representations with global structural information. In: CIKM '15, pp. 891–900. ACM, New York (2015)
12. Chen, C., Tong, H.: Fast eigen-functions tracking on dynamic graphs. In: Proceedings of the 2015 SIAM International Conference on Data Mining, pp. 559–567. SIAM (2015)
13. Clement, P., Desch, W.: An elementary proof of the triangle inequality for the wasserstein metric. Proc. Am. Math. Soc. **136**(1), 333–339 (2008)
14. Courty, N., Flamary, R., Ducoffe, M.: Learning wasserstein embeddings (2017). arXiv preprint arXiv:1710.07457
15. Courty, N., Flamary, R., Tuia, D., Rakotomamonjy, A.: Optimal transport for domain adaptation. IEEE Trans. Pattern Anal. Mach. Intell. **39**(9), 1853–1865 (2017)

16. Cuturi, M., Doucet, A.: Fast computation of wasserstein barycenters. In: International Conference on Machine Learning, pp. 685–693 (2014)
17. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Math. Control Signal Syst. **2**(4), 303–314 (1989)
18. Dash, N.S.: Context and contextual word meaning. SKASE J. Theor. Linguist. **5**(2), 21–31 (2008)
19. De Goes, F., Breeden, K., Ostromoukhov, V., Desbrun, M.: Blue noise through optimal transport. ACM Trans. Graph. (TOG) **31**(6), 171 (2012)
20. Delalleau, O., Bengio, Y., Roux, N.L.: Efficient non-parametric function induction in semi-supervised learning. In: AISTATS '05, pp. 96–103 (2005)
21. Doersch, C.: Tutorial on variational autoencoders (2016). arXiv preprint arXiv:1606.05908
22. Dreyfus, S.: The numerical solution of variational problems. J. Math. Anal. Appl. **5**(1), 30–45 (1962)
23. Eom, Y.-H., Jo, H.-H.: Tail-scope: using friends to estimate heavy tails of degree distributions in large-scale complex networks. Sci. Rep. **5** (2015)
24. Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., Bengio, S.: Why does unsupervised pre-training help deep learning? J. Mach. Learn. Res. **11**, 625–660 (2010)
25. Givens, C.R., Shortt, R.M., et al.: A class of wasserstein metrics for probability distributions. Mich. Math. J. **31**(2), 231–240 (1984)
26. Glorot, X., Bordes, A., Bengio,Y.: Deep sparse rectifier neural networks. In: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pp. 315–323 (2011)
27. Grover, A., Leskovec, J.: Node2vec: scalable feature learning for networks. In: KDD '16, pp. 855–864. ACM, New York (2016)
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition (2015). arXiv preprint arXiv:1512.03385
29. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.-R., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al.: Deep neural networks for acoustic modeling in speech recognition: the shared views of four research groups. IEEE Signal Process. Mag. **29**(6), 82–97 (2012)
30. Hinton, G.E., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. Neural Comput. **18**(7), 1527–1554 (2006)
31. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
32. Holland, P.W., Leinhardt, S.: Holland and Leinhardt reply: some evidence on the transitivity of positive interpersonal sentiment (1972)
33. Hornik, K.: Approximation capabilities of multilayer feedforward networks. Neural Netw. **4**(2), 251–257 (1991)
34. Jamali, M., Ester, M.: A matrix factorization technique with trust propagation for recommendation in social networks. In: Proceedings of the Fourth ACM Conference on Recommender Systems, pp. 135–142. ACM (2010)
35. Jin, E.M., Girvan, M., Newman, M.E.: Structure of growing social networks. Phys. Rev. E **64**(4), 046132 (2001)
36. Kingma, D., Ba, J.: Adam: a method for stochastic optimization (2014). arXiv preprint arXiv:1412.6980
37. Kolouri, S., Park, S.R., Thorpe, M., Slepcev, D., Rohde, G.K.: Optimal mass transport: signal processing and machine-learning applications. IEEE Signal Process. Mag. **34**(4), 43–59 (2017)
38. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
39. LeCun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
40. LeCun, Y., Chopra, S., Hadsell, R., Ranzato, M., Huang, F.: A tutorial on energy-based learning. Predict. Struct. Data **1** (2006)

41. Leicht, E.A., Holme, P., Newman, M.E.: Vertex similarity in networks. Phys. Rev. E **73**(2), 026120 (2006)
42. Liben-Nowell, D., Kleinberg, J.: The link-prediction problem for social networks. J. Am. Soc. Inf. Sci. Technol. **58**(7), 1019–1031 (2007)
43. Luo, D., Nie, F., Huang, H., Ding, C.H.: Cauchy graph embedding. In: Proceedings of the 28th International Conference on Machine Learning (ICML-11), pp. 553–560 (2011)
44. Ma, J., Cui, P., Zhu, W.: Depthlgp: learning embeddings of out-of-sample nodes in dynamic networks. In: AAAI, pp. 370–377 (2018)
45. Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Eleventh Annual Conference of the International Speech Communication Association, pp. 1045–1048 (2010)
46. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119 (2013)
47. Nathan, E., Bader, D.A.: A dynamic algorithm for updating katz centrality in graphs. In: Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017, pp. 149–154. ACM (2017)
48. Ou, M., Cui, P., Pei, J., Zhang, Z., Zhu, W.: Asymmetric transitivity preserving graph embedding. In: Proceedings of ACM SIGKDD, pp. 1105–1114 (2016)
49. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab (1999)
50. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: online learning of social representations. In: SIGKDD, pp. 701–710. ACM (2014)
51. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press (2005)
52. Rossi, R.A., Ahmed, N.K.: Role discovery in networks. IEEE Trans. Knowl. Data Eng. **27**(4), 1112–1131 (2015)
53. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Neurocomputing: foundations of research. In: Learning Representations by Back-Propagating Errors, pp. 696–699. MIT Press, Cambridge (1988)
54. Salakhutdinov, R., Hinton, G.: Semantic hashing. Int. J. Approx. Reason. **50**(7), 969–978 (2009)
55. Shaw, B., Jebara, T.: Structure preserving embedding. In: Proceedings of the 26th Annual International Conference on Machine Learning, pp. 937–944. ACM (2009)
56. Siegelmann, H.T., Sontag, E.D.: On the computational power of neural nets. J. Comput. Syst. Sci. **50**(1), 132–150 (1995)
57. Smola, A.J., Kondor, R.: Kernels and Regularization on Graphs, pp. 144–158. Springer, Berlin (2003)
58. Socher, R., Perelygin, A., Wu, J.Y., Chuang, J., Manning, C.D., Ng, A.Y., Potts, C.: Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), vol. 1631, pp. 1642. Citeseer (2013)
59. Stoyanov, V., Ropson, A., Eisner, J.: Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In: AISTATS'11, Fort Lauderdale, April 2011
60. Sun, L., Ji, S., Ye, J.: Hypergraph spectral learning for multi-label classification. In: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 668–676. ACM (2008)
61. Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., Mei, Q.: Line: large-scale information network embedding. In: Proceedings of the 24th International Conference on World Wide Web, pp. 1067–1077. International World Wide Web Conferences Steering Committee (2015)
62. Tenenbaum, J.B., De Silva, V., Langford, J.C.: A global geometric framework for nonlinear dimensionality reduction. Science **290**(5500), 2319–2323 (2000)
63. Tian, F., Gao, B., Cui, Q., Chen, E., Liu, T.-Y.: Learning deep representations for graph clustering. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, pp. 1293–1299 (2014)

64. Tolstikhin, I., Bousquet, O., Gelly, S., Schoelkopf, B.: Wasserstein auto-encoders (2017). arXiv preprint arXiv:1711.01558
65. Tu, K., Cui, P., Wang, X., Wang, F., Zhu, W.: Structural deep embedding for hyper-networks. In: Thirty-Second AAAI Conference on Artificial Intelligence, pp. 426–433 (2018)
66. Tu, K., Cui, P., Wang, X., Yu, P.S., Zhu, W.: Deep recursive network embedding with regular equivalence. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2357–2366. ACM (2018)
67. Vilnis, L., McCallum, A.: Word representations via Gaussian embedding (2014). arXiv preprint arXiv:1412.6623
68. Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M.: Graph kernels. J. Mach. Learn. Res. **11**, 1201–1242 (2010)
69. Wang, D., Cui, P., Zhu, W.: Structural deep network embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1225–1234. ACM (2016)
70. Wang, H., Banerjee, A.: Bregman alternating direction method of multipliers. In: Advances in Neural Information Processing Systems, pp. 2816–2824 (2014)
71. Werbos, P.J.: Backpropagation through time: what it does and how to do it. Proc. IEEE **78**(10), 1550–1560 (1990)
72. Zang, C., Cui, P., Faloutsos, P., Zhu, W.: Long short memory process: modeling growth dynamics of microscopic social connectivity. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 565–574. ACM (2017)
73. Zhu, D., Cui, P., Wang, D., Zhu, W.: Deep variational network embedding in wasserstein space. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 2827–2836. ACM (2018)
74. Zhu, X., Ghahramani, Z.: Learning from labeled and unlabeled data with label propagation. Technical report (2002)
75. Zhu, X., Ghahramani, Z., Lafferty, J.: Semi-supervised learning using Gaussian fields and harmonic functions. In: ICML'03, pp. 912–919. AAAI Press (2003)
76. Zhuang, J., Tsang, I.W., Hoi S.: Two-layer multiple kernel learning. In: International Conference on Artificial Intelligence and Statistics, pp. 909–917 (2011)

Deep Neural Networks for Corrupted Labels

Ishan Jindal, Matthew Nokleby, Daniel Pressel, Xuewen Chen and Harpreet Singh

Abstract The success of deep convolutional networks on image and text classification and recognition tasks depends on the availability of large, correctly labeled training datasets, but obtaining the correct labels for these gigantic datasets is very difficult task. To deal with this problem, we describe an approach for learning deep networks from datasets corrupted by unknown label noise. We append a nonlinear noise model to a standard deep network, which is learned in tandem with the parameters of the network. Further, we train the network using a loss function that encourages the clustering of training images. We argue that the non-linear noise model, while not rigorous as a probabilistic model, results in a more effective denoising operator during backpropagation. We evaluate the performance of proposed approach on image classification task with artificially injected label noise to MNIST, CIFAR-10, CIFAR-100 and ImageNet datasets and on a large-scale Clothing 1M dataset with inherent label noise. Further, we show that with the different initialization and the regularization of the noise model, we can apply this learning procedure to text classification tasks as well. We evaluate the performance of modified approach on TREC text classification dataset. On all these datasets, the proposed approach provides significantly improved classification performance over the state of the art and is robust

I. Jindal (✉) · M. Nokleby · X. Chen · H. Singh
Wayne State University, Detroit, MI, USA
e-mail: ishan.jindal@wayne.edu

M. Nokleby
e-mail: matthew.nokleby@wayne.edu

X. Chen
e-mail: xuewen.chen@wayne.edu

H. Singh
e-mail: hsingh1@wayne.edu

D. Pressel
Interactions Digital Roots, Ann Arbor, MI, USA
e-mail: dpressel@interactions.com

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_7

to the amount of label noise and the training samples. This approach is computationally fast, completely parallelizable, and easily implemented with existing machine learning libraries.

Keywords Label noise · Deep learning · Image classification · Text classification · E-M style label denoising · Convolutional network

1 Introduction

The last decade has seen dramatic advances in image classification, image captioning, object recognition, and more, owing mostly to deep convolutional neural networks (CNNs) trained on large, labeled datasets [1–4]. Researchers often benchmark the performance of these algorithms on standard, curated datasets such as MNIST, CIFAR, ImageNet, or MSCOCO [2, 5–7]. However, use of curated data sets elides a crucial point: in practical datasets, labels are not always reliable. "Crowdsourced" labels obtained from social media or other non-expert sources are subject to error, and in subjective tasks even humans or experts may disagree on the correct label. (For a taxonomy of types and sources of label noise, see [8] and the references therein.) As deep learning systems become more complex and are trained on even more massive datasets, it becomes increasingly difficult to obtain clean labels. In this scenario, an approach to learning that accounts for noisy labels is needed.

Zhu and Wu [9] perform an extensive study on the effect of label noise on classification performance of a classifier and find that noise in input features is less important than noise in training labels.

In this work, we present a two-pronged approach for image classification tasks, as shown in Fig. 1, to learning CNNs from training sets corrupted by label noise having unknown statistics. The first prong is to augment the CNN architecture with a nonlinear model for the label noise, which is learned during training. A challenge with

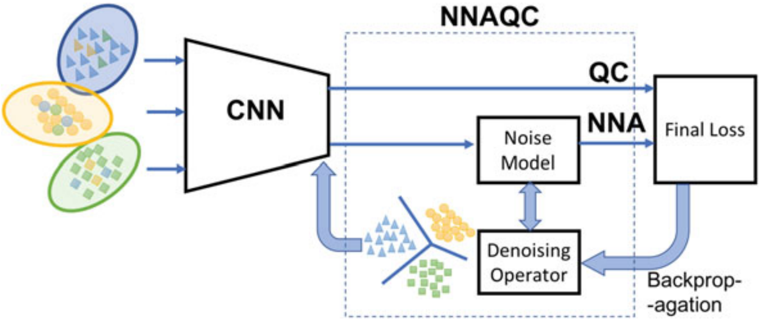


Fig. 1 Proposed architecture for learning deep CNNs from noisy labels. The learned noise model act as a denosing operator while backpropagation

this approach is that the model is underdetermined: when CNN outputs disagree with the labels provided, it is not clear whether the CNN should update its weights to match the noisy labels or update the noise model to account for the possibility of incorrect labels. The nonlinear model turns out to be particularly effective in handling these situations. We show that the proposed model is actually non-rigorous as a transition probability between clean and noisy labels; however, it results in a "denoising" operator that better handles errors when training the CNN via backpropagation. Because the noise model is not used at test time, learning an accurate and rigorous noise model is less important than the impact of the noise model on CNN training. The second prong is to augment the standard cross-entropy loss with a term that encourages the CNN to cluster images in feature space. This allows the network to learn from the natural clustering of the data, even when labels are unreliable.

We demonstrate the performance of the proposed approach on the MNIST, CIFAR-10, CIFAR-100 and ImageNet datasets corrupted by label noise. On these datasets, the proposed approach exhibits state-of-the-art performance in terms of classification accuracy over a clean test set. The results show that the proposed approach is scalable to a large number of image categories. The results are robust to label noise, achieving near-optimum performance when there is little noise, and maintaining classification accuracy as the label noise increases. We emphasize that this robustness does not require knowledge of the label noise statistics or tuning of hyperparameters. The performance of the proposed approach degrades gracefully as the training size decreases, suggesting that it sufficiently regularizes the learning of the combined noise and classification model. Indeed, in some cases the performance is *better* when using fewer training samples, which suggests that the sample complexity of the model is occasionally too high for the given dataset, in which case regularization in the form of early stopping improves performance. Finally, we evaluate the proposed approach on Clothing 1M [10] dataset, consisting of 1M images with noisy labels.

For the text classification task, the noise model prong is sufficient and we study the effect of different initialization, regularization, and batch sizes when training with noisy labels. We observe that proper initialization and regularization helps the noise model learn to be robust to even extreme amounts of noise. Finally, we use low-dimensional projections of the features of the training examples to understand the effectiveness of the noise model.

The rest of the paper is organized as follows. First, we describe the label nosie in detail in Sect. 2, and then we review the existing literature on this subject in Sect. 3. Then, we describe the details of the proposed approach and its different components in Sect. 4. Finally, in Sect. 5 we present experimental results on variety of image and text classification datasets and conclude our work with future directions.

2 Label Noise

In label noise, the label of a training sample is swapped with the other label. We define the *label flip* as when the label of a training sample is swapped with another label within the dataset and the *outlier*, when the label of a training sample is swapped with another label which is not present in the dataset. Label noise is explained in Fig. 2, where the top row shows the training samples, middle row provides the true labels associated with each image (in this work we assume that this knowledge is not available to the classifier) and the last row represents the supplied unreliable training labels, where red text denotes the label flips, green text denotes the correct labels and highlighted red text denotes the outliers. These label swaps can be of different types, for instance, it can be uniform, random or class-conditional label swaps. In this work, we study the effects of uniform and random label flips on the performance of the standard deep network.

The extraordinary performance of deep learning methods heavily depends on the availability of tons of training data points. Though it is easy to obtain such a huge amount of data points, it is challenging to get the associated correct labels. However, getting labels for these many training samples is a very laborious task and often prone to label noise such as labels flips, mislabeling, and outliers. One easy and least expensive way is to get the annotations using search engines and/or from social media sites. This yields to the poor quality of annotations and these poor quality annotations are the source of label noise.

Though standard deep networks are robust to label noise to some extent, say 10–15% of label noise, the performance degrades when the noise percentage is higher than 15% as shown in Fig. 3. We plot the classification accuracy with respect to the increasing label noise percentage in training dataset, where blue and orange bars represents the classification accuracy when the labels are altered according to uniform and random label noise respectively. For this experiment we alter the labels of the Trec dataset according to uniform and random label noise and train a standard text deep neural network model [11] (more information in Sect. 5 end-to-end. From this plot, we can easily observe that the label noise has deteriorating effect on the

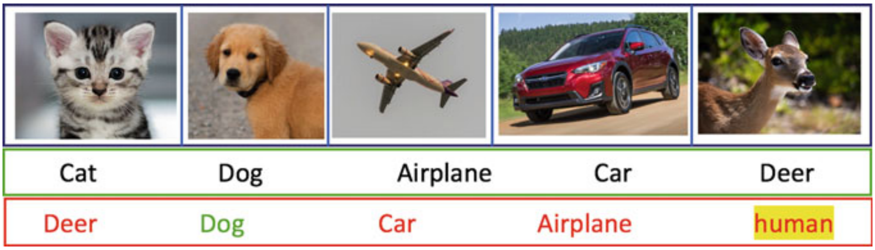


Fig. 2 Training datasets with corrupted labels

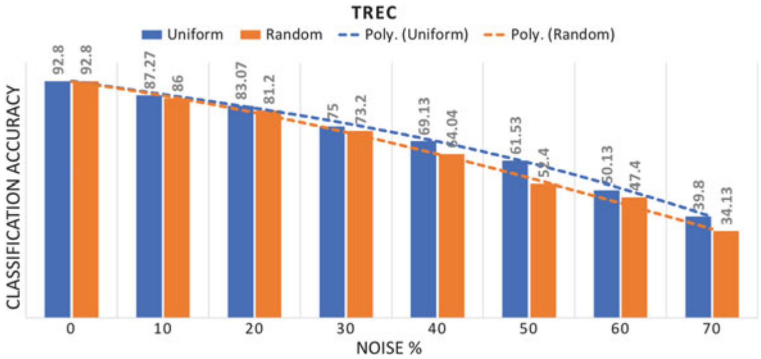


Fig. 3 Classification results for Trec text classification dataset

performance of deep network and a significant performance drop can be seen at 70% label noise. Also, random label flipping further degrade the performance of the network.

3 Relationship to Prior Work

This problem is closely related to semi-supervised and weakly-supervised learning, for which there is an extensive body of work. We refer the reader to [12] for survey.

Previous work addresses the question of learnability when labels are binary and label noise is i.i.d. and class-independent [13], and provides sample complexity bounds in terms of the VC dimension for the 0–1 loss. More recently, [14] provides sample complexity bounds for more general loss functions, in terms of the Rademacher complexity, for *class-conditional* label noise having known statistics. The upshot of these works is that if labels are flipped with probability η, the sample complexity increases roughly by a factor of $1/(1-2\eta)^2$. Equivalently, the generalization error scales roughly as $1/(\sqrt{n}(1-2\eta))$ instead of the usual $\sqrt{1/n}$. Other possible solution to this problem includes estimation of the noise rate. A class conditional estimator for estimating the noise rate is proposed in [15].

Earlier works consider label noise for general learning algorithms. For example, [16] presents a method for learning a kernel-based classifier from noisy labels with unknown statistics. Using an EM-style algorithm, their approach learns jointly a generative noise model and a classifier. Similarly, [17] employs an EM-style algorithm to estimate the reliability of labels. Other techniques detect and discard samples with anomalous labels [18] or relabel erroneous samples [19]. In a similar vein to [14], a recent work shows that careful choice of the loss function leads to learning that is provably robust to label noise [20].

Recently, authors have begun designing CNNs to deal with label noise for image classification [21, 22]. In some of the approaches [23–25], a standard CNN is augmented with a generative noise model that must be learned in tandem with the CNN parameters. A joint optimization framework presented in [26] simultaneously learns the parameters and estimates the true labels. As mentioned above, the augmented model is underdetermined and must be regularized, else the network may choose the identity as the noise transition matrix. Each of these works imposes a different regularization term to encourage a non-trivial noise model: [23] imposes a cost on the trace of the label noise transition probability matrix, whereas [24] uses Dropout regularization. In [27], an unified distillation framework is proposed to learn CNN from the noisy labels. This framework uses label relations in knowledge graphs and a small clean dataset to learn a classifier from noisy labels. Reference [28] proposes to train in parallel two neural networks, which weights are updated only when label predictions disagree.

A number of works have attempted to address this problem of learning from corrupted labels for deep networks. These approaches can be divided into two categories; attempts to mitigate the effect of label noise using auxiliary clean data, and attempts to learn directly from the noisy labels.

Presence of auxiliary clean data: This line of research exploits a small, clean dataset to correct the corrupted labels. For instance, [27] learn a teacher network with clean data to re-weight a noisy label with a soft label in the loss function. Similarly, [29] use the clean data as a label correction network. One can use this auxiliary source of information to do inference over latent clean labels [30]. Further, [31] models the auxiliary trustworthiness of noisy image labels to alleviate the effect of label noise. Though these methods show very promising results, the absence of clean data in some situations might hinder the applicability of these methods.

Learning directly from noisy labels: This research directly learns from the noisy labels by designing a robust loss function, or by modeling the latent labels. For instance, [32], apply bootstrapping to the loss function to have consistent label prediction for similar images. In a similar vein, [33] identifies and discards outliers in order to fine-tune a pretrained CNN. Similarly, [34] alleviate the label noise effect by adequately weighting the loss function using the sample number. Reference [35] propose a sequential meta-learning model that takes in a sequence of loss values and outputs the weights for the labels. Reference [36] further explores the conditions on loss functions such that the loss function is noise tolerant.

A number of approaches learn the transition from latent labels to the noisy labels. For example, [37] propose a noise adaptation framework for symmetric label noise. Based on this work, several other works [23, 24, 38, 39] account for the label noise by learning a noisy layer on top of a DNN where the learned transition matrix represents the label flip probabilities. Similar to [14, 38] estimates a noise model in a theoretical motivated manner during a pre-training phase of the network, and then correct the loss function. It estimates the transition matrix heuristically but with the large number of classes this estimation is not easy to obtain. On CIFAR-10, [23] provides competitive performance as long as the label noise is not too strong. Similarly, [10] propose a probabilistic image conditioned noise model. Reference [33] proposed an image regularization technique to detect and discard the noisy labeled images. Other approaches include building two parallel classifiers [40] where one classifier deals with image recognition and the other classifier models humans reporting bias.

The proposed approach incorporates the spirit of [23, 24]—in that it learns an explicit noise model—and clusters the data via an EM-style approach. The intuition is to identify mislabeled images by their inconsistency with similar images, which combats label noise and emulates unsupervised learning. The upshot of this work is that a label noise model is beneficial, especially when it is regularized by an unsupervised component in the loss function. However, learning a correct noise model is neither necessary nor sufficient for state-of-the-art performance. Indeed, the proposed approach uniformly outperforms a genie-aided CNN, similar to [24]. The proposed approach denoise the gradient of the loss by a denoising operator before being fed into the gradient of the base model parameters. Our approach produces a very diffuse denoising operator and thus prevents the base model from learning the noisy label directly.

4 Proposed Approach

We consider the supervised learning of a classifier of d-dimensional images that belong to one of L image classes. Let the (noise-free) training set be denoted by

$$\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\},$$

where $x_i \in \mathbb{R}^d$ is the ith image and $y_i \in \{1, \ldots, L\}$ is its label, and where implicitly there is an unknown joint distribution $p(x, y)$ on the image/label pairs. Ideally, one would train a classifier on the training set $\mathcal{D}$, but we suppose that instead of access to the noise-free training set $\mathcal{D}$, we obtain a training set with unreliable labels. Let this *noisy* training set be denoted by

$$\mathcal{D}' = \{(x_1, y_1'), (x_2, y_2'), \ldots, (x_n, y_n')\},$$

where y_i' is a potentially erroneous label for x_i. We suppose *class-conditional* label noise, where the noisy label y_i' depends only on the true label y_i, but not on the image x_i or any other labels y_j or y_j'. Under this model, the label noise is characterized by the conditional distribution $p(y'|y)$, which we describe via the $L \times L$ column-stochastic matrix ϕ, with

$$\phi_{ij} = p(y' = j | y = i).$$

We use a noise model parameterized by the overall probability of a label error, denoted by $0 \leq p \leq 1$:

$$\phi = (1 - p)\mathbf{I} + p\Delta, \tag{1}$$

where $\mathbf{I}$ is the identity matrix, and Δ is a matrix with zeros along the diagonal and remaining entries of each column are drawn uniformly and independently from the $L-1$-dimensional unit simplex. That is, the label error probability for each class is p, while the probability distribution *within* the erroneous classes is drawn uniformly at random.

Our objective is to train a CNN, using the noisy set $\mathcal{D}'$, that makes accurate predictions of the true label y given an input image x. It is straightforward to train a CNN that predicts the *noisy* labels. The conditional distribution for the noisy label of the image x can be written as:

$$p(y' = \hat{y}'|x) = \sum_{i} p(y' = \hat{y}'|y = \hat{y}_i)p(y = \hat{y}_i|x). \tag{2}$$

One can learn the classifier associated with $p(y' = \hat{y}'|x)$ via standard training on the noisy set $\mathcal{D}'$. To predict the clean labels, i.e. to learn the conditional distribution $p(y = \hat{y}_i|x)$ requires more effort, as we cannot extract the "clean" classifier from the noisy classifier when the label noise distribution is unknown.

4.1 Proposed Approach

The architecture of our proposed framework is shown in Fig. 4. We take a standard deep CNN—which we call the *base model*—and augment it with a model that accounts for the label noise. The base and noise models are trained jointly using $\mathcal{D}'$ via stochastic gradient descent. The noise model is used only during training, in which it effectively "denoises" the gradients associated with the noisy labels during backpropagation in order to improve the learning of the base model. Because there is no need to predict the noisy labels of test images, we disconnect the noise model at test time and classify using the base model alone.

We stack an additional one fully-connected processing layer[1] on top of base CNN model. We lump the base model parameters—processing layer weights and biases, etc.—into a parameter vector Θ. The high-level features that the base model outputs, which we denote via $t_1(x; \Theta) \in \mathbb{R}^L$, are put through the usual softmax function:

$$\sigma(z)_i = \frac{\exp(z_i)}{\sum_{j=1}^{L} \exp(z_i)}$$

to produce the conditional distribution of the clean label; i.e. $p(y|x; \Theta) = \sigma(t_1(x; \Theta))$, from which we can predict the clean label of an image x. A distinct feature of the proposed approach is that we use a *nonlinear* transformation between the estimate of the clean labels and the estimate of the noisy labels. In an abuse of

[1] We emphasize that the proposed framework can be applied to any CNN architecture.

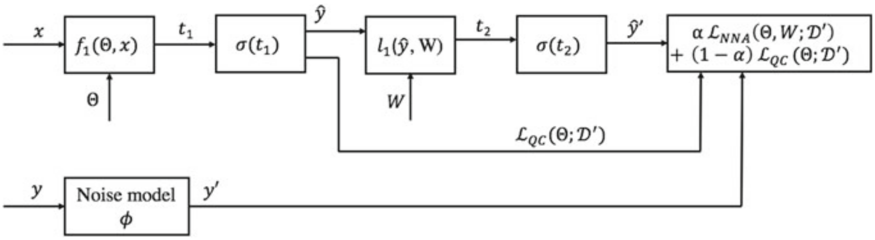

x
$f_1(\Theta, x)$
Θ
t_1
$\sigma(t_1)$
$\hat{y}$
$l_1(\hat{y}, W)$
W
t_2
$\sigma(t_2)$
$\hat{y}'$
$\alpha \mathcal{L}_{NNA}(\Theta, W; \mathcal{D}')$
$+ (1-\alpha) \mathcal{L}_{QC}(\Theta; \mathcal{D}')$
$\mathcal{L}_{QC}(\Theta; \mathcal{D}')$
y
Noise model
ϕ
y'

Fig. 4 The proposed framework

notation, let $\hat{y} = \sigma(t_1)$ denote the probabilities $p(y|x; \Theta)$. To obtain the probabilities of the noisy labels, denoted $\hat{y}'$, we perform a softmax regression on $\hat{y}$:

$$p(y'|x; \Theta, W) := \sigma(W\sigma(t_1(x; \Theta))), \tag{3}$$

where $W \in \mathbb{R}^{L\times L}$ is a square matrix that governs the transition probabilities. We emphasize a subtle point: This formulation is not rigorous probabilistically. Equation (3) does *not* correctly compute marginal probability of the noisy labels according to (2). To be consistent with the law of total probability, we *should* calculate conditional distribution as

$$p(y' = i|y = j; W) = [\sigma(We_j)]_i, \tag{4}$$

where e_j is the jth elementary vector. From this conditional distribution, the distribution on y' should be

$$p(y'|x; \Theta, W) = \sigma(W)\sigma(t_1(x; \Theta)), \tag{5}$$

where $\sigma(W)$ is the softmax function applied to each column of W. This is equivalent to the architecture used in [23], where a simple linear layer with column-stochastic weight matrix is learned, ideally to match the matrix ϕ that governs the label noise. By taking a nonlinear transformation of $\hat{y}$, instead of a linear transformation associated with transition probabilities, we violate the laws of probability in computing $\hat{y}'$. Nevertheless, empirically we see that the resulting classifier has excellent performance, and in the next part we give a justification for this approach.

Now, the challenge is to learn jointly the CNN parameters Θ and the nonlinear noise model parameters W. One approach is to minimize the standard cross-entropy loss of the end-to-end model, which we call the *nonlinear noise-aware* loss $\mathcal{L}_{\text{NNA}}$:

$$\begin{aligned}\mathcal{L}_{\text{NNA}}(\Theta, W; \mathcal{D}') &= -\frac{1}{n}\sum_{i=1}^{n} \log p(y' = \hat{y}_i' | x_i; \Theta, W) \\ &= -\frac{1}{n}\sum_{i=1}^{n} \log[\sigma(W\sigma(t_1(x_i; \Theta)))]_{\hat{y}_i}.\end{aligned}$$

Empirically we see that this loss function leads to quite good predictions of the true labels. However, $\mathcal{L}_{\text{NNA}}$ does not directly encourage the model to predict correctly the true label $\hat{y}$ as the true label; instead, the prediction of $\hat{y}$ is judged only indirectly via the noisy label predictions $\hat{y}'$. Indeed, this approach treats $\hat{y}$ as an additional hidden layer that acts as an information bottleneck.

To encourage good predictions of $\hat{y}$, we need to feed $\hat{y}$ into the loss function directly. To do so, we introduce an additional term that encourages a "quasi-clustering" of the training images. Images that are close in feature space usually will have the same label, a fact that we can exploit when dealing with noisy labels. We penalize the cross-entropy between a linear combination of the predicted labels and the noisy labels and the predicted labels themselves, i.e.

$$\begin{aligned}\mathcal{L}_{\text{QC}}(\Theta; \mathcal{D}') &= -\frac{1}{n}\sum_{i=1}^{n} \left(\beta p(\hat{y}_i | x_i; \Theta) + (1-\beta) y_i'\right) \times \log p(\hat{y}_i | x_i; \Theta), \\ &= -\frac{1}{n}\sum_{i=1}^{n} (\beta[\sigma(t_1(x; \Theta))]_{\hat{y}_i} + (1-\beta) y_i') \times \log[\sigma(t_1(x; \Theta))]_{\hat{y}_i}.\end{aligned}$$

This type of loss function has been used widely in the literature, such as in [32, 41, 42], and it has the effect of clustering the data. For a large value of β, minimizing this loss function encourages $\hat{y}$ toward a low-entropy vector, i.e. one with most of its mass on a single point. In order to make such confident predictions, the CNN needs to map similar output features to similar classes, which is equivalent to clustering. Finally, we form the *nonlinear, noise-aware, quasi-clustering* loss, denoted $\mathcal{L}$, by taking a convex combination of the two losses:

$$\mathcal{L}_{\text{NNAQC}}(\Theta, W; \mathcal{D}') = \alpha\mathcal{L}_{\text{NNA}}(\Theta, W; \mathcal{D}') + (1-\alpha)\mathcal{L}_{\text{QC}}(\Theta; \mathcal{D}'), \tag{6}$$

and we minimize the NNAQC loss via standard back-propagation over the noisy training set $\mathcal{D}'$. We obtain the values of α and β via cross-validation.

For the text classification task, we find that the NNA step alone provides the state of the art performance and we explore the effect of different initialization of the noise model on the classification performance along with different regularizations. We use the following loss function to train our network for text classification datasets.

$$\mathcal{L}_{\text{text}}(\Theta, W; \mathcal{D}') = -\frac{1}{n}\sum_{i=1}^{n} \log[\sigma(W\sigma(t_1(x_i; \Theta)))]_{\hat{y}_i} + \frac{1}{2}\lambda ||W||_2^2. \tag{7}$$

4.2 Justifying the Nonlinear Noise Model

In this section, we study the effect of the proposed nonlinear noise model. At first instance, it seems that we are violating the basic laws of probability by adding a nonlinear softmax layer at the output as described above. We emphasize, however, that the role of the noise model is *not* to make accurate predictions of the noisy labels, but to encourage the learning of a CNN that makes accurate predictions of the clean labels instead of noisy ones. Therefore, the ultimate test of a noise model is the extent to which it improves training. To that end, the proposed architecture is designed not to learn an explicit noise model, but to learn a "denoising" operator that effectively filters the gradients associated with the noisy labels. To see the benefits of this approach, we examine the back-propagation gradient steps for the base model parameters for the proposed architecture and for a CNN augmented with a standard linear noise model.

In Fig. 5, we zoom in on the proposed approach architecture. For an input sample x, we lump all the initial convolutional, ReLu and pooling layers into one function $f_1(\Theta, x)$ with parameters Θ, and we obtain the normalized prediction of true labels $\hat{y}$ via the first softmax layer $\sigma(t_1)$. We pass the clean label predictions through the matrix W and take the softmax function to obtain the noise label distribution $\hat{y}'$. To observe the effect of the prediction $\hat{y}$ and the noise model parameters W on the learning process, we write down the gradient of the loss function $\mathcal{L}$ with respect to Θ:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \frac{\partial \mathcal{L}}{\partial \hat{y}'} \frac{\partial \sigma(t_2)}{\partial t_2} \frac{\partial l_1(\hat{y}, W)}{\partial \hat{y}} \frac{\partial \sigma(t_1)}{\partial t_1} \frac{\partial f_1(\Theta, x)}{\partial \Theta} \tag{8}$$

$$= \frac{\partial \mathcal{L}}{\partial \hat{y}'} \left(\frac{\partial \sigma(t_2)}{\partial t_2} W \frac{\partial \sigma(t_1)}{\partial t_1} \right) \frac{\partial f_1(\Theta, x)}{\partial \Theta}. \tag{9}$$

For comparison, in Fig. 6 we consider a linear noise model as described in (4) and (5), where the matrix W determines the noise model via the stochastic matrix $\sigma(W)$. We write the gradient steps similar to the previous case as:

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \frac{\partial \mathcal{L}}{\partial \hat{y}'} \frac{\partial l_1(\hat{y}, \sigma(W))}{\partial \hat{y}} \frac{\partial \sigma(t_1)}{\partial t_1} \frac{\partial f_1(\Theta, x)}{\partial \Theta} \tag{10}$$

$$= \frac{\partial \mathcal{L}}{\partial \hat{y}'} \left(\sigma(W) \frac{\partial \sigma(t_1)}{\partial t_1} \right) \frac{\partial f_1(\Theta, x)}{\partial \Theta}. \tag{11}$$

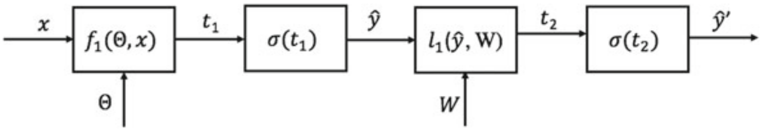


Fig. 5 Augmented linear layer with Softmax

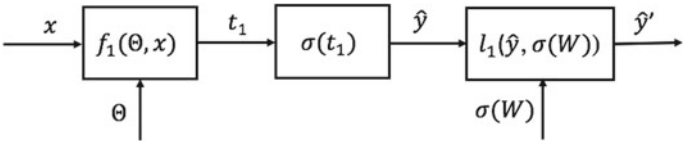


Fig. 6 No softmax augmentation

Comparing (9) with (11) reveals a few crucial points. First of all, in each case the gradient of the loss $\partial\mathcal{L}/\partial\hat{y}'$ is "denoised" by an operator before being fed into the gradient of the base model parameters. This is the main role of the noise model: to prevent the base model from learning the noisy labels directly. In the case of the proposed approach, the denoising operator is $\frac{\partial\sigma(t_2)}{\partial t_2}\, W\, \frac{\partial\sigma(t_1)}{\partial t_1}$, and in the case of the linear noise model, the denoising operator is $\sigma(W)\, \frac{\partial\sigma(t_1)}{\partial t_1}$.

Second, we find that the proposed approach denoising operator is more diffuse than the linear noise model. To see this, consider an estimate $\hat{y}$ that places most of the probability on a single class. In the proposed approach, the resulting noisy label prediction is $\hat{y}' = \sigma(W\hat{y})$; applying the softmax to $W\hat{y}$ "spreads out" the probabilities, and the prediction of the noisy label will be less concentrated on a single class than the equivalent linear model. In other words, the nonlinear noise model is intrinsically less confident than a rigorous linear model. Because the denoising operator contains the term $\frac{\partial\sigma(t_2)}{\partial t_2}$, which is a function of $\sigma(W\hat{y})$, the resulting operator is more diffuse, i.e. its columns are less concentrated on individual values.

A more diffuse operator allows for more flexibility in handling disagreements between the CNN model predictions and the noisy labels. Consider the case in which the CNN outputs a prediction $\hat{y}$ concentrated around a single value (say, i) that is different than the (perhaps erroneous) training label y' (say, j). Here, the challenge is to decide whether y' is an error or whether the CNN prediction is bad. In a linear noise model, the denoising operator has most of its weight concentrated on the ith row. On the other hand, the loss gradient $\partial\mathcal{L}/\partial\hat{y}'$ has all of its weight on the jth row. Therefore, the denoising operator wipes out most of the gradient, and the result is largely to ignore the sample. With the proposed approach, the denoising operator is not as concentrated around row i, so the backpropagation step attempts to learn more from the training point, even though the model prediction and noisy label disagree.

Similarly, the proposed approach prevents the model from being overconfident when the model and noisy label agree. If the CNN makes a confident prediction $\hat{y}$ and $i = j$, the combination of a non-diffuse denoising operator and the gradient $\partial\mathcal{L}/\partial\hat{y}'$ has large-magnitude elements, and the model is overconfident is supposing that the label is not noisy. The diffuse denoising operator resulting from the proposed approach, on the other hand, spreads out the gradient, preventing an over-aggressive backpropagation step. To sum up, the proposed approach denoising operator encourages the CNN to learn from a training sample when there is disagreement, and discourages overfitting when there is agreement. Finally, the quasi-clustering regularization in our approach in Fig. 4 provides information to base model about the true labels by clustering all the samples that are close in feature space. We also write the backpropagation gradient steps with quasi-clustering regularization as

$$\frac{\partial \mathcal{L}}{\partial \Theta} = \gamma \frac{\partial \sigma(t_1)}{\partial t_1} \frac{\partial f_1(\Theta, x)}{\partial \Theta} \tag{12}$$

where,

$$\gamma = \left(\frac{\partial \mathcal{L}}{\partial \hat{y}'} \frac{\partial \sigma(t_2)}{\partial t_2} W\right) + \frac{\partial \mathcal{L}'_{\mathrm{QC}}}{\partial \hat{y}}.$$

We also consider the learning performance when the noise model ϕ is known exactly. One might expect that learning a base CNN using a linear noise model, with the transition matrix set at ϕ, would provide superior performance. Somewhat surprisingly, [24] reports cases where even this "genie-aided" approach is outperformed. We observe the similar behavior; augmenting the CNN with the true noise model performs significantly worse than the proposed approach. As suggested by the above analysis, the nonlinear noise model simply results in a more effective denoising operator, even when the model does not learn the underlying noise statistics.

5 Experimental Results

In this Section, we evaluate the empirical performance of NNAQC on variety of different image classification and text classification datasets and compare it with other approaches.

5.1 General Setting

Image Classification Tasks In all the experiments, we use the MATLAB toolbox MatConvNet [43]. We evaluate the performance of the classifier on the MNIST [44], CIFAR-10, CIFAR-100 [6], ImageNet [7] and Clothing 1M [10] datasets. We use the default CNN architectures and parameters provided in MatConvNet for CIFAR-10 and MNIST datasets as a base CNN model. For other datasets, we use the CNN architectures that provide the best classification accuracy on corresponding clean datasets and use it as a base CNN. We provide the details of these architectures in subsequent sections. For all the NNAQC experiments we use $\alpha = 0.9$ and $\beta = 0.9$. We compare NNAQC to several other algorithms: a standard, noise-ignorant CNN trained on $\mathcal{D}'$ ("base model"); a CNN augmented with the true noise model ϕ ("genie-aided"); the genie-aided model using the quasi-clustering loss function ("genie-aided with QC"); the dropout-regularized model of [24] ("dropout"); the trace-regularized model of [23] ("trace"); the soft bootstrapping algorithm of [32] ("bootstrapping"); and the forward loss correction of [38] ("F-correction"). We also try adding dropout regularization to the noise model of NNAQC (“regu.NNAQC”). For an apples-to-apples comparison we fixed the base model for all the approaches and implement their methods on top of it. In all the experiments we train CNN end-to-end via stochastic gradient descent method with batch size 100. For CIFAR-10 and MNIST datasets, we run the experiment 5 times for each setting and report the mean.

Text Classification Tasks For text classification experiments, we use a publicly-available deep learning library *Baseline*—a fast model development tool for NLP tasks [45]. We choose a commonly-used, high-performance model from [11] as a base model and train according to (7). To examine the robustness of the proposed approach, we intentionally flip the class labels with 0–70% label noise, in other words: $p \in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7\}$, and observe the effect of different types of label flipping, such as uniform (*Uni*) and random (*Rand*) label flipping, along with instance-dependent label noise. For all the experiments, we use early stopping based on validation set accuracy where the class labels in validation are also corrupted.

We use the different name convention to show the effect of different regularizes on the classification performance. We indicate the performance of a standard deep network *Without Noise model (WoNM)* on the noisy label dataset. We also plot the results for the stacked *Noise Model Without Regularization (NMWoRegu)* and stacked *Noise Model With Regularization (NMwRegu)*. Unless otherwise stated, in all the deep networks with the stacked noise model, we initialize the noise layer parameters as an identity matrix. We further analyze the effect of the noise layer initialization on the overall performance. We define *TDwRegu* as the stacked noise model with regularization, initialized with true injected noise distribution and *RandwRegu* as the stacked noise model with regularization, initialized randomly. We run all experiments five times and report the mean accuracy.

5.2 Artificial Label Noise

To examine the robustness of NNAQC on artificially injected noise, we corrupt the true labels according to (1) with $p \in \{0, 0.05, 0.10, 0.30, 0.50, 0.70\}$.

CIFAR-10: We train our CNN on CIFAR-10 dataset [6], a subset of 80 million Tiny Image dataset [3]. It contains natural images of size 32 × 32 × 3 from 10 different categories. It has 50 K training and 10 K test images. On the *clean* dataset, the base model CNN achieves 20.49% classification error. We produce a noisy dataset $\mathcal{D}'$ by corrupting the labels according to the noise distribution (1) for each value of p. Table 1 first row shows the comparative performance of NNAQC when the networks are trained using 50 K training samples. In all cases, NNAQC, perhaps regularized by dropout, substantially outperforms other approaches. This includes the genie-aided approaches, bolstering our claim that it is less important to know the noise statistics than to learn an effective denoising operator for training. Further, NNAQC is robust to variations in the noise level, recovering near-optimum performance when

Table 1 NNAQC performance for different datasets and compared to other approaches w.r.t number of training samples

	Training samples	50 K					
	Noise %	0	5	10	30	50	70
CIFAR-10	Base model	**20.49**	23.00	25.30	30.49	39.47	65.60
	Genie-aided	20.50	21.07	24.32	28.09	39.29	62.38
	Genie-aided (QC)	20.98	22.22	23.23	25.52	33.98	57.57
	Trace	22.48	23.00	23.90	27.20	39.06	63.00
	Bootstrapping	23.33	23.76	25.00	28.64	35.07	66.14
	Dropout	37.29	36.90	31.30	25.40	31.28	63.04
	F-correction	21.00	21.45	22.10	23.70	29.12	58.91
Ours	NNAQC	21.11	21.85	**22.03**	24.20	28.41	56.12
	Regu. NNAQC	20.96	**21.40**	22.05	**23.10**	**28.06**	**56.09**
	Training samples	60 K					
	Noise %	0	5	10	30	50	70
MNIST	Base model	**00.89**	02.67	03.68	04.50	34.50	48.80
	Genie-aided	00.89	02.67	03.68	04.50	34.50	48.80
	Trace	01.29	01.40	01.46	02.12	03.80	24.20
	Bootstrapping	01.29	01.30	01.41	02.00	03.60	22.20
	Dropout	01.29	01.29	01.32	01.83	02.83	24.60
	F-correction	01.12	01.13	01.19	01.50	02.23	21.00
Ours	NNAQC	01.14	01.15	01.24	01.83	02.20	**16.42**
	Regu. NNAQC	01.01	**01.08**	**01.18**	**01.46**	**02.19**	18.70

there is little noise. Although, we notice that NNAQC performances better than NNAQC with dropout regularization (Regu. NNAQC) in some of the cases, but this performance gap is negligible. However, we observe a significant performance gap with the datasets having more than 10 classes.

To evaluate the robustness of NNAQC with respect to varying training dataset size, in Table 1, we show the performance of all the approaches as a function of number of training samples. For every dataset, we starts with original number of training samples and keep on decrease the samples by 20K, as shown in Table 1. For CIFAR-10 dataset, in column 1 we train all the models with all 50 K training samples. We also find that the performance of *F-correction* [38] is close to the performance to the NNAQC, however, as we reduce the training dataset size NNAQC outperforms F-correction significantly. Since [38] works by estimating the noise transition matrix, the performance gap on smaller training set further strengthens our claim that learning a correct noise model is neither necessary nor sufficient for state-of-the-art performance in the presence of label noise.

We also compare NNAQC to [33], which uses a pre-trained AlexNet to obtain high level features for training images and fine-tunes a final softmax layer on $\mathcal{D}'$. Because they use a pre-trained network where NNAQC and other approaches train a CNN from scratch, a direct comparison of results is impossible. However, in the presence of 50% noise for 50 K training samples, [33] reports 28% classification error rate, compared to 28.41% for NNAQC. That is, NNAQC performs competitively with this approach even though it is not pre-trained, which may indicate that it is a more powerful approach overall.

MNIST: We perform similar experiments on handwritten digits dataset MNIST [44], which contains 60 K training images of the 10 digits of size 28 × 28 and 10 K test images. We produce a noisy dataset $\mathcal{D}'$ as in the CIFAR-10 case. On the clean dataset, the base model CNN achieves a classification error rate of 0.89%. In Table 1 (Last row) we again see that NNAQC provides superior performance overall and is robust to both high noise power and a smaller training set. Similar to CIFAR-10, we compare the performance of NNAQC against the pre-trained/fine-tuned strategy [33] on the MNIST dataset. In the presence of 50% noise NNAQC outperforms the [33], achieving 2.2% classification error while [33] achieves at minimum 7.63% classification error.

CIFAR-100: We next show the performance of NNAQC on a dataset with more classes, making the problem more challenging: CIFAR-100 [6] which consists of 32 × 32 color images of 100 different categories containing 600 images each. There are 500 images for training and 100 images for testing per class. Because of the complexity of this dataset, we use a different base CNN model with two conv+ReLU+max pool layers, two FC layers and a softmax layer. This is a low capacity CNN network

Table 2 NNAQC performance on CIFAR-100 with different CNN architectures and compared to other approaches

	Noise %	0	5	10	30	50	60
LC-CNN	Base model	**50.90**	52.48	53.82	60.38	68.46	88.20
	Trace	53.12	54.27	55.00	58.70	64.50	84.12
	Bootstrapping	54.20	54.90	55.30	59.00	69.75	88.30
	Dropout	65.80	63.54	62.01	57.76	63.24	84.19
	F-correction	56.68	57.13	57.11	62.67	66.12	83.90
Ours	NNAQC	52.31	52.40	53.10	56.68	63.00	84.00
	Regu. NNAQC	52.29	**52.33**	**53.00**	**56.91**	**62.20**	**83.13**
	Noise %	0	5	10	30	50	60
44-layer ResNet	Base model	**30.99**	31.54	33.86	36.50	64.60	84.89
	Trace	31.56	31.50	34.10	36.00	65.41	84.82
	Bootstrapping	31.60	31.50	34.06	36.32	63.45	84.30
	Dropout	55.20	53.04	52.13	37.68	64.11	85.00
	F-correction	31.00	31.13	**33.12**	35.80	61.24	**84.00**
Ours	NNAQC	31.00	31.14	34.01	35.88	61.35	85.00
	Regu. NNAQC	31.12	**31.13**	33.16	**35.71**	**61.20**	84.03

(LC-CNN) with a classification error rate of 50.9% on the clean dataset. In order to verify that the robustness of NNAQC is not due to the low capacity models, we also evaluate NNAQC on a high capacity deep residual network (ResNet) [46] with 30.99% classification error rate on clean labels. We use ResNet with depth 44 and the same training parameters as described in [38].

We compare the NNAQC performance on CIFAR-100 in Table 2. Here we train the networks on entire training data. Similar to previous experiments we fixed the base model CNN for all the approaches. In Table 2 (First row), we show the competitive performance of NNAQC over other approaches when trained on LC-CNN. We observed that the performance of NNAQC on CIFAR-100 is consistent with MNIST and CIFAR-10, proves the scalability of NNAQC. Here, dropout particularly improves performance (Regu. NNAQC), likely because the larger label noise model benefits from regularization. We also show the performance of NNAQC on ResNet architecture in Table 2 (Second row). We observe that among other approaches only F-correction performs equally well with NNAQC at a number of occasions, however, with the LC-CNN the scenario is different—NNAQC performs better than all the other approaches. Comparing NNAQC performance on ResNet with LC-CNN, it is clear that the NNAQC performance is independent of base CNN network architecture. This claim is further strengthened by our experiments on Clothing 1M datasets with different CNN architectures in the next section.

ImageNet: We further test the scalability of NNAQC to a 1000 class classification problem. We show the performance of NNAQC on ImageNet 2012 dataset [7] which has 1.3M image with clean labels over 1000 categories. For this experiment, we use CNN model of Krizhevsky et al. [1] as the base model. This CNN model has five conv+RELU+max pool layers, two FC layers and a softmax layer. As described in [23], we generate a column stochastic noise distribution matrix (ϕ) such that for a particular class, noise is randomly distributed to only 10 other randomly chosen classes. For 50% label noise, each class has 50% correct labels and other 50% labels are randomly distributed among 10 randomly chosen classes. Since our main intention here is to show the scalability of NNAQC to a large number of classes and to maintain the simplicity, we transfer the parameters of first four convolutional blocks from a pre-trainined AlexNet model. While training, we keep the parameters of first four convolutional blocks (conv+ReLU+max pool) intact/frozen and only train the last convolutional block, two FC layers, a softmax layer and the stacked NNAQC layer. In Table 3 we compare the NNAQC performance with the base Alexnet model (i.e. no noise model) on the validation set images, with 0, 10, and 50%, randomly-distributed corrupted labels. We observe a slight performance gain for NNAQC over the base model with "clean" labels-perhaps due to label noise inherent in the ImageNet dataset. We observe that the 50% label noise significantly hurts the performance of the base model whereas the NNAQC withstands and shows a superior performance (a clear gain of ∼11.0%) over the base model. Here, dropout regularization (Regu. NNAQC) further improves the overall performance by 2.51%.

Table 3 ImageNet validation set classification error rate

	Noise %	Top 5 val. error		
		0	10	50
ImageNet	Base model	19.20	31.21	53.46
	Trace	19.10	29.00	46.24
Ours	NNAQC	19.30	29.10	44.31
	Regu. NNAQC	**18.30**	**28.21**	**41.80**

TREC[2] Reference [47] is a question classification dataset consisting of fact based questions divided into broad semantic categories. We use a six-class version of TREC dataset. For this dataset, the base model network architecture consists of an input and embedding layer + [3] one feature windows with 100 feature maps and dropout rate 0.5 with batch size 10.

We evaluate the performance of our model on TREC dataset in Table 4 in the presence of uniform and random label noise and compare the performance with the base model (*WoNM*) as our baseline. In all the regimes, the proposed approach is significantly better than the baseline for both random and uniform label noise. For all datasets, we observe a gain of approximately 30% w.r.t the baseline in the presence of extreme label noise. We do observe a drop in classification accuracy as we increase the percentage of label noise but even at the extreme label noise our method outperformed the baseline method. Interestingly, if we assume an oracle to determine prior knowledge of true noise distribution (*TDwRegu01*), it does not necessarily improve classification performance, especially for multi-class classification problems. In addition to this, we also observe a slight performance gain for the proposed approach over the baseline with clean labels—perhaps due to label noise inherent in the datasets.

5.3 *Real Label Noise*

Finally, we evaluate the performance of NNAQC on real world noisy label dataset Clothing 1M [10] in terms of classification error rate. This dataset contains 1M images with noisy labels from 14 different classes. Along with the incorrectly labeled images, this dataset provides 50 K clean images for training; 14 k for validation; and 10 k for testing. For this dataset, we use a 50-layer ResNet pre-trained on ImageNet dataset as a base model. Similar to [38], we train the network with different weight-decay parameter depending on the training dataset size. In Table 5 we compare the performance of NNAQC with a number of existing approaches.

At first, we see a clear performance improvement of ∼3% with ResNet in comparison to AlexNet (#1 vs. #3). On clean training images NNAQC (#7) performs better than the base model (#3) as expected. On noisy images with ImageNet pre-

[2] http://cogcomp.cs.illinois.edu/Data/QA/QC/.

Table 4 Test performance on TREC text classification dataset

		Batch size	**10**														
		Label flips	*Uniform*							*Random*							
TREC	Noise%	Clean data	10	20	30	40	50	60	70	0	10	20	30	40	50	60	70
	WoNM(%)	92.8	87.6	83.6	75.87	67.27	57.4	46.27	42.8	92.8	85.93	82.2	74.0	68.4	53.53	48.2	31.47
	TDwRegu01(%)	50.87	45.33	45.4	36.33	25.87	28.33	16.87	16.87	50.87	56.4	36.8	24.0	25.47	22.6	18.8	22.6
	NMWoRegu(%)	92.33	88.07	84.67	76.4	68.47	58.4	50.07	41.33	92.07	85.87	84.27	72.47	66.53	50.13	44.6	33.0
	NMwRegu001(%)	92.47	90.53	88.07	81.6	73.47	64.07	55.87	43.67	92.4	88.53	86.4	77.2	67.67	54.67	47.93	**34.87**
	NMwRegu01(%)	**92.73**	**90.8**	**89.53**	**88.67**	**84.93**	**79.67**	**69.67**	**52.4**	**92.7**	**90.33**	**90.6**	**86.47**	**83.07**	**70.93**	**65.2**	33.4
		Batch size	**50**														
		Label flips	*Uniform*							*Random*							
	Noise%	Clean labels	10	20	30	40	50	60	70	0	10	20	30	40	50	60	70
	WoNM(%)	92.8	87.27	83.07	75.00	69.13	61.53	50.13	39.8	92.8	86.00	81.2	76.2	64.07	52.4	47.4	34.13
	TDwRegu01(%)	55.73	50.4	44.73	39.6	22.27	25.67	14.93	21.00	55.73	45	44.93	27.73	27.87	22.6	17.87	22.6
	NMWoRegu(%)	92.6	87.73	83.33	76.33	70.67	56.8	48.2	39.67	92.60	85.27	83.00	73.6	65.8	50.4	45.93	30.73
	NMwRegu001(%)	92.53	90.73	87.20	82.53	73.93	65.07	52.87	44.60	92.53	88	87.2	79.07	71.2	51.67	49.00	33.40
	NMwRegu01(%)	**92.53**	**91.33**	**90.27**	**88.47**	**83.87**	**77.87**	**68.73**	**55.67**	**92.53**	**90.00**	**90.2**	**85.93**	**82.6**	**71.4**	**67.33**	**37.53**

Table 5 NNAQC performance on Clothing 1M dataset. #10 shows the best results. #6 is reported results from [26]

Clothing1M				
#	Model/method	Init	Training	Error
1	AlexNet/cross-	ImageNet	50 k	28.17
2	AlexNet/trace	#1	1 M, 50 k	24.84
3	50-ResNet/cross-	ImageNet	50 K	25.12
4	50-ResNet/F-corr-	ImageNet	1 M	30.16
5	50-ResNet/cross-	#4	50 k	19.62
6	50-ResNet/[26]	ImageNet	1 M	27.77
7	50-ResNet/NNAQC	ImageNet	50 K	25.10
8	50-ResNet/NNAQC	ImageNet	1 M	27.73
9	50-ResNet/NNAQC	ImageNet	1 M, 50 K	24.58
10	50-ResNet/cross-	#8	50 K	**19.45**

training, we gain a 3% performance improvement compared to F-correction. Also, in comparison to a very recent work [26] (#6 vs. #8), NNAQC performance is very competitive. Further, we observe the effect of availability of clean 50k images on the NNAQC performance, that is, given the clean labels, NNAQC performance improved by ∼3% (#8 vs. #9). In a similar vein to [38], we first train NNAQC on 1M noisy images (#8) and fine tune the network with 50 k clean images (#10), we observe that the NNAQC outperforms all the methods in Table 5 and is very competitive overall.

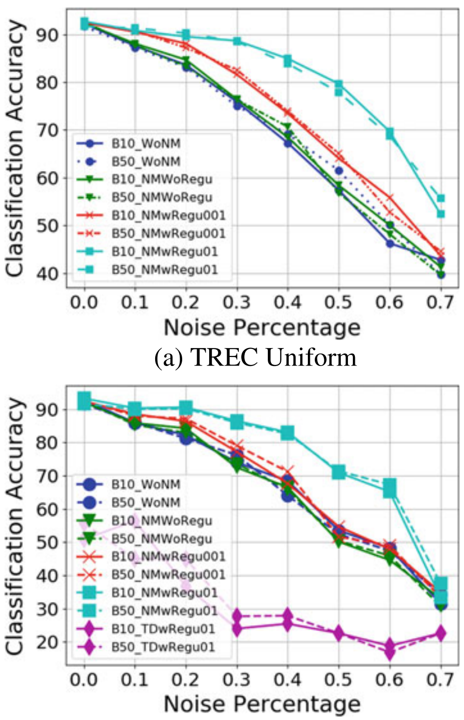
5.4 *Effect of Batch Size*

We also observe the effect of different batch sizes on performance as described in [48]. For all datasets, we do observe small performance gains for highly non-uniform noisy labels, for instance 70%, in Fig. 7 row 2. However, for uniform label flips, we do not observe performance gains with increasing batch size.

Table 6 SVM classification

	TRB			TRPr		
Data(N%)	WoNM	Noisy	True	NMwRegu01	Noisy	True
SST2 (40)	70.24	70.95	79.24	82.32	73.90	83.25
AG (70)	59.70	52.44	79.18	90.33	86.27	89.4
AG (60)	83.25	68.8	88.28	90.45	87.77	90.78
TREC (40)	66.80	63.4	79.0	73.40	69.6	83.2
TREC (20)	83.6	80.0	86.0	87.40	83.6	90.0

Fig. 7 Effect of batch size on label noise classification for different datasets. [Best viewed in color]



(a) TREC Uniform

(b) TREC Random

5.5 *Understanding Noise Model*

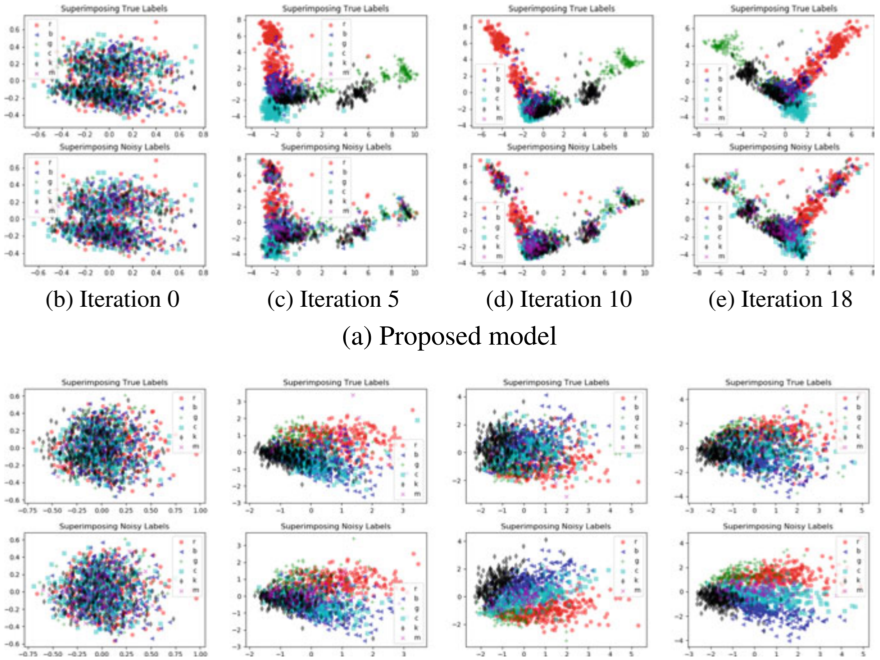
In order to further understand the noise model, we first train the base model and the proposed model on noisy labels. Afterward, we collect the last fully-connected layer's activations for all the training samples and treat them as the learned feature representation of the input sentence. We get two different sets of feature representations, one corresponding to the base model (*TRB*), and the other corresponding to the proposed model (*TRPr*). Given these learned feature representations—the artificially injected noisy labels and the true labels of the training data—we learn two different SVMs for each model, with and without noise. For the base model, for both SVMs, we use TRB representation as inputs and train the first SVM with the true labels as targets and the second SVM with the unreliable labels as targets. Similarly, we train two SVMs for the proposed model. After training, we evaluate the performance of all the learned SVMs on clean test data in Table 6, where the 1st column represents the corresponding model performance, "Noisy" and "True" column represents the SVM performance when trained on noisy and clean labels, respectively. We run these experiments for different datasets with different label noise.

The SVM, trained on TRB and noisy labels, is very close to the base model performance (6). This suggests that the base model is just fitting the noisy labels.

On the other hand, when we train an SVM on the TRPr representations with true labels as targets, the SVM achieves the proposed model performance. This means that the proposed approach helps the base model to learn better feature representations even with the noisy targets, which suggest that this noise model is learning a label denoising operator.

We analyze the representation of training samples in feature domain by plotting the t-SNE embeddings [49] of the TRB and TRPr. For brevity, we plot the t-SNE visualizations for TREC dataset with 50% label noise in Fig. 8.

For each network, we show two different t-SNE plots. For example in Fig. 8a we plot two rows of t-SNE embeddings for the proposed model. In the first row of Fig. 8a, each training sample is represented by its corresponding true label, while in the second row (the noisy label plot) each training sample is represented by its corresponding noisy label. We observe that, as the learning process progresses, the noise model helps the base model to cluster the training samples in the feature domain. With each iteration, we can see the formation of clusters in Row 1. However, in Row



(b) Iteration 0 (c) Iteration 5 (d) Iteration 10 (e) Iteration 18

(a) Proposed model

(g) Iteration 0 (h) Iteration 5 (i) Iteration 10 (j) Iteration 18

(f) No noise model stacked

Fig. 8 t-SNE visualization of the last layer activations of a base network before softmax for TREC Dataset with 50% corrupted labels; First row in (a) when the corresponding true labels are superimposed on the t-SNE data points; Second row in (a) when the noisy labels are superimposed onto the t-SNE data points. [Best viewed in color]

2, when the noisy labels are superimposed, the clusters are not well separated. This means that the noise model denoises the labels and presents the true labels to the base network to learn.

In Fig. 8f, we plot two rows of t-SNE embeddings of the TRB representations. It seems that the network directly learns the noisy labels. This provides further evidence to support [50]'s finding that the deep network memorizes data without knowing of true labels. In Row 2 of Fig. 8f, we can observe that the network learns noisy features representations which can be well clustered according to given noisy labels.

6 Conclusion and Future Work

In this work we describe a scalable and effective approach towards training a deep networks on noisy data labels. We show the performance of this approach on variety of different datasets with different noise regimes and varying training data sizes for different modalities. We observe that this approach is model agnostic and can be applied to any deep architecture. We augmented a standard deep neural network with a non-linear noise model that models the label noise. The capabilities of this noise model are further enhanced by adding an extra unsupervised component to the final loss function. To learn the classifier and the noise model jointly, we apply different regularization to the weights of the final softmax layer. One way to interpret the results of this approach is that the deep network is encouraged to learn to cluster the data–rather than to classify it–to a greater extent than one would expect from the noise statistics. In other words, it is better to let deep networks cluster ambiguously-labeled data than to risk learning noisy labels. The details of this phenomenon–including which noise model is "ideal" for training an accurate network–is a topic for future research. Further, we anticipate that this model can handle instance dependent label noise as well, that is, quasi clustering step accounts for instance-dependent noise without learning a full instance-dependent noise model. Future works shall consider analyzing the instance dependent label noise.

References

1. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems, pp. 1097–1105 (2012)
2. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: common objects in context. In: European Conference on Computer Vision, pp. 740–755. Springer (2014)
3. Torralba, A., Fergus, R., Freeman, W.T.: 80 million tiny images: a large data set for nonparametric object and scene recognition. IEEE Trans. pattern Anal. Mach. Intell. **30**(11), 1958–1970 (2008)
4. Johnson, J., Karpathy, A., Fei-Fei, L.: Densecap: Fully convolutional localization networks for dense captioning. In: IEEE Conference on Proceedings of the Computer Vision and Pattern Recognition, pp. 4565–4574 (2016)

5. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
6. Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images (2009)
7. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: a large-scale hierarchical image database. In: IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2009, pp. 248–255. IEEE (2009)
8. Frénay, B., Verleysen, M.: Classification in the presence of label noise: a survey. IEEE Trans. Neural Netw. Learn. Syst. **25**(5), 845–869 (2014)
9. Zhu, X., Wu, X.: Class noise vs. attribute noise: a quantitative study. Artif. Intell. Rev. **22**(3), 177–210 (2004)
10. Xiao, T., Xia, T., Yang, Y., Huang, C., Wang, X.: Learning from massive noisy labeled data for image classification. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2691–2699 (2015)
11. Kim, Y.: Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1746–1751 (2014)
12. Zhu, X.: Semi-supervised learning literature survey (2005)
13. Aslam, J.A., Decatur, S.E.: On the sample complexity of noise-tolerant learning. Inf. Process. Lett. **57**(4), 189–195 (1996)
14. Natarajan, N., Dhillon, I.S., Ravikumar, P.K., Tewari, A.: Learning with noisy labels. In: Advances in Neural Information Processing Systems, pp. 1196–1204 (2013)
15. Liu, T., Tao, D.: Classification with noisy labels by importance reweighting. IEEE Trans. Pattern Anal. Mach. Intell. **38**(3), 447–461 (2016)
16. Lawrence, N.D., Schölkopf, B.: Estimating a kernel fisher discriminant in the presence of label noise. In: ICML, vol. 1, Citeseer, pp. 306–313 (2001)
17. Rebbapragada, U., Brodley, C.E.: Class noise mitigation through instance weighting. In: European Conference on Machine Learning, pp. 708–715. Springer (2007)
18. Brodley, C.E., Friedl, M.A., et al.: Identifying and eliminating mislabeled training instances. In: AAAI/IAAI, vol. 1, pp. 799–805 (1996)
19. Brodley, C.E., Friedl, M.A.: Identifying mislabeled training data. J. Artif. Intell. Res. **11**, 131–167 (1999)
20. Manwani, N., Sastry, P.: Noise tolerance under risk minimization. IEEE Trans. Cybern. **43**(3), 1146–1151 (2013)
21. Ma, X., Wang, Y., Houle, M.E., Zhou, S., Erfani, S.M., Xia, S.T., Wijewickrema, S., Bailey, J.: Dimensionality-driven learning with noisy labels (2018). arXiv:1806.02612
22. Wang, Y., Liu, W., Ma, X., Bailey, J., Zha, H., Song, L., Xia, S.T.: Iterative learning with open-set noisy labels. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 8688–8696 (2018)
23. Sukhbaatar, S., Bruna, J., Paluri, M., Bourdev, L., Fergus, R.: Training convolutional networks with noisy labels (2014). arXiv:1406.2080
24. Jindal, I., Nokleby, M., Chen, X.: Learning deep networks from noisy labels with dropout regularization. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 967–972. IEEE (2016)
25. Jindal, I., Pressel, D., Lester, B., Nokleby, M.: An effective label noise model for dnn text classification (2019). arXiv:1903.07507
26. Tanaka, D., Ikami, D., Yamasaki, T., Aizawa, K.: Joint optimization framework for learning with noisy labels. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5552–5560 (2018)
27. Li, Y., Yang, J., Song, Y., Cao, L., Luo, J., Li, L.J.: Learning from noisy labels with distillation. In: ICCV, pp. 1928–1936 (2017)
28. Malach, E., Shalev-Shwartz, S.: Decoupling "when to update" from "how to update". In: Advances in Neural Information Processing Systems, pp. 961–971 (2017)
29. Veit, A., Alldrin, N., Chechik, G., Krasin, I., Gupta, A., Belongie, S.: Learning from noisy large-scale datasets with minimal supervision. In: The Conference on Computer Vision and Pattern Recognition (2017)

30. Vahdat, A.: Toward robustness against label noise in training deep discriminative neural networks. In: Advances in Neural Information Processing Systems, pp. 5596–5605 (2017)
31. Yao, J., Wang, J., Tsang, I.W., Zhang, Y., Sun, J., Zhang, C., Zhang, R.: Deep learning from noisy image labels with quality embedding. IEEE Trans. Image Process. (2018)
32. Reed, S., Lee, H., Anguelov, D., Szegedy, C., Erhan, D., Rabinovich, A.: Training deep neural networks on noisy labels with bootstrapping (2014). arXiv:1412.6596
33. Azadi, S., Feng, J., Jegelka, S., Darrell, T.: Auxiliary image regularization for deep cnns with noisy labels (2015). arXiv:1511.07069
34. Joulin, A., van der Maaten, L., Jabri, A., Vasilache, N.: Learning visual features from large weakly supervised data. In: European Conference on Computer Vision, pp. 67–84. Springer (2016)
35. Jiang, L., Zhou, Z., Leung, T., Li, L.J., Fei-Fei, L.: Mentornet: Regularizing very deep neural networks on corrupted labels (2017). arXiv:1712.05055
36. Ghosh, A., Kumar, H., Sastry, P.: Robust loss functions under label noise for deep neural networks. In: AAAI, pp. 1919–1925 (2017)
37. Mnih, V., Hinton, G.E.: Learning to label aerial images from noisy data. In: Proceedings of the 29th International conference on machine learning (ICML-12), pp. 567–574 (2012)
38. Patrini, G., Rozza, A., Menon, A.K., Nock, R., Qu, L.: Making deep neural networks robust to label noise: a loss correction approach. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition(CVPR), pp. 2233–2241 (2017)
39. Han, B., Yao, J., Niu, G., Zhou, M., Tsang, I., Zhang, Y., Sugiyama, M.: Masking: A new perspective of noisy supervision (2018). arXiv:1805.08193
40. Misra, I., Lawrence Zitnick, C., Mitchell, M., Girshick, R.: Seeing through the human reporting bias: visual classifiers from noisy human-centric labels. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2930–2939 (2016)
41. Goldberger, J., Ben-Reuven, E.: Training deep neural-networks using a noise adaptation layer (2017)
42. Audhkhasi, K., Osoba, O., Kosko, B.: Noise-enhanced convolutional neural networks. Neural Netw. **78**, 15–23 (2016)
43. Vedaldi, A., Lenc, K.: Matconvnet: convolutional neural networks for matlab. In: Proceedings of the 23rd ACM international conference on Multimedia, pp. 689–692. ACM (2015)
44. LeCun, Y., Cortes, C., Burges, C.J.: The MNIST database of handwritten digits (1998)
45. Pressel, D., Ray Choudhury, S., Lester, B., Zhao, Y., Barta, M.: Baseline: a library for rapid modeling, experimentation and development of deep learning algorithms targeting nlp. In: Proceedings of Workshop for NLP Open Source Software (NLP-OSS), Association for Computational Linguistics, pp. 34–40 (2018)
46. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778 (2016)
47. Voorhees, E.M., Tice, D.M.: The TREC-8 question answering track evaluation. In: TREC, vol. 82, (1999)
48. Rolnick, D., Veit, A., Belongie, S., Shavit, N.: Deep learning is robust to massive label noise (2017). arXiv:1705.10694
49. Van Der Maaten, L.: Accelerating t-sne using tree-based algorithms. J. Mach. Learn. Res. **15**(1), 3221–3245 (2014)
50. Zhang, C., Bengio, S., Hardt, M., Recht, B., Vinyals, O.: Understanding deep learning requires rethinking generalization (2016). arXiv:1611.03530

Constructing a Convolutional Neural Network with a Suitable Capacity for a Semantic Segmentation Task

Yalong Jiang and Zheru Chi

Abstract Although the state-of-the-art performance has been achieved in many computer vision tasks such as image classification, object detection, saliency prediction and depth estimation, Convolutional Neural Networks (CNNs) still perform unsatisfactorily in some difficult tasks such as human parsing which is the focus of our research. The inappropriate capacity of a CNN model and insufficient training data both contribute to the failure in perceiving the semantic information of detailed regions. The feature representations learned by a high-capacity model cannot generalize to the variations in viewpoints, human poses and occlusions in real-world scenarios due to overfitting. On the other hand, the under-fitting problem prevents a low-capacity model from developing the representations which are sufficiently expressive. In this chapter, we propose an approach to estimate the complexity of a task and match the capacity of a CNN model to the complexity of a task while avoiding under-fitting and overfitting. Firstly, a novel training scheme is proposed to fully explore the potential of low-capacity CNN models. The scheme outperforms existing end-to-end training schemes and enables low-capacity models to outperform models with higher capacity. Secondly, three methods are proposed to optimize the capacity of a CNN model on a task. The first method is based on improving the orthogonality among kernels which contributes to higher computational efficiency and better performance. In the second method, the convolutional kernels within each layer are evaluated according to their semantic functions and contributions to the training and test accuracy. The kernels which only contribute to the training accuracy but has no effect on the testing accuracy are removed to avoid overfitting. In the third method, the capacity of a CNN model is optimized by adjusting the dependency among convolutional kernels. A novel structure of convolutional layers is proposed to reduce the

Y. Jiang (✉) · Z. Chi
Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong
e-mail: AllenYLJiang@outlook.com; yalong.jiang@connect.polyu.hk

Z. Chi
e-mail: enzheru@polyu.edu.hk

Z. Chi
Hong Kong Polytechnic University Shenzhen Research Institute, Shenzhen, China

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_8

number of parameters while maintaining the similar performance. Besides capacity optimization, we further propose a method to evaluate the complexity of a human parsing task. An independent CNN model is trained for this purpose using the labels for pose estimation. The evaluation on complexity is achieved based on estimated pose information in images. The proposed scheme for complexity evaluation was conducted on the Pascal Person Part dataset and the Look into Person dataset which are for human parsing. The schemes for capacity optimization were conducted on our models for human parsing which were trained on the two data sets. Both quantitative and qualitative results demonstrate that our proposed algorithms can match the capacity of a CNN model well to the complexity of a task.

Keywords Convolutional neural networks (CNNs) · Under-fitting · Over-fitting · Capacity optimization · Complexity evaluation

1 Introduction

The research works in deep learning have mostly focused on building models with wider [1] or deeper [2, 3] architectures to achieve better performance in applications. Only a few papers have paid attention to the necessary capacity required for a CNN to be competent for a task. For instance, it is addressed in [4] that the capacity of a machine learning model measures how complex a function it can model. A model with a higher capacity can represent more complex relationships between variables. Moreover, [5] defined the capacity of a model as the logarithm of the number of functions it can implement. To evaluate whether a ReLU-based neural network is competent for a task, firstly the process of inference should be equalized to parametrically mapping the high-dimensional distributions in datasets to a latent space. The distributions of images are represented by a polyhedral manifold which is partitioned into pieces (cells) and different pieces are mapped into the latent space independently. The rectified linear complexity of the manifold of images and that of a neural network are compared to decide whether a CNN model can encode the data [5]. The former is evaluated by the minimal number of pieces required to piecewisely map images to latent spaces and is determined by the distribution of images. The latter is determined by the upper bound of the number of piecewise functions that can be implemented by a CNN. A CNN is competent for a task only when its rectified linear complexity surpasses the complexity of the distribution of images. Similarly, VC dimension, the growth function, the Rademacher and Gaussian complexity, the metric entropy, and the minimum description length (MDL) have been proposed to evaluate the complexity of a task and the capacity of a model.

Although the above-mentioned papers have conducted analysis on CNNs' capacity, their conclusions can only be applied to preliminary tasks such as CIFAR-10 [6]. Different from the works mentioned above, we have tried to match the capacity of models to that of high-level vision tasks such as image segmentation. Our chapter is divided into four parts. Firstly, the performance comparison on some general models are given in Introduction to show the advantage of capacity optimization. The comparison demonstrates the possibility of simplifying CNNs while maintaining performance. Secondly, we propose the method to better explore the potential of CNNs to improve performance without increasing complexity. Thirdly, our work on estimating the complexity of a segmentation task is introduced. Finally, methods of matching the capacity of a CNN model to the complexity of a tasks are proposed. Novel training strategies, schemes for data augmentation, improved architectures of deep learning models and analysis on datasets are all included in the chapter. As a result, this chapter is closely related to the architectural design of deep learning models, one of the major focuses of this volume.

Performance achieved by high-capacity models and low-capacity models. For a CNN, the number of piecewise mapping functions can be measured by the number of independent convolutional kernels that can transform the inputs in different ways. Different kernels within the same layer extract complementary cues. Kernels in higher layers allow for different compositions of the outputs from lower layers.

The architectures of CNNs have evolved for years with performance and capacity increased significantly. For instance, the accuracy on the ImageNet dataset [7] has been significantly improved by structures such as AlexNet [8], VGG [9], GoogLeNet [10], ResNet [11], DenseNet [12], ResNeXt [13], SE-Net [14] and the automatically designed architectures such as those reported in [15–17]. Moreover, the residual connections [11] and batch normalization [18] have made it possible to build extremely deep CNNs with more than 1000 layers [19]. It is shown in [5] that deeper CNNs have higher capacity. Figure 1 shows the relation between the accuracy on the ImageNet challenge [7] and the capacity of a CNN. The CNNs in Fig. 1 are with the same architecture but differ in capacity and depth.

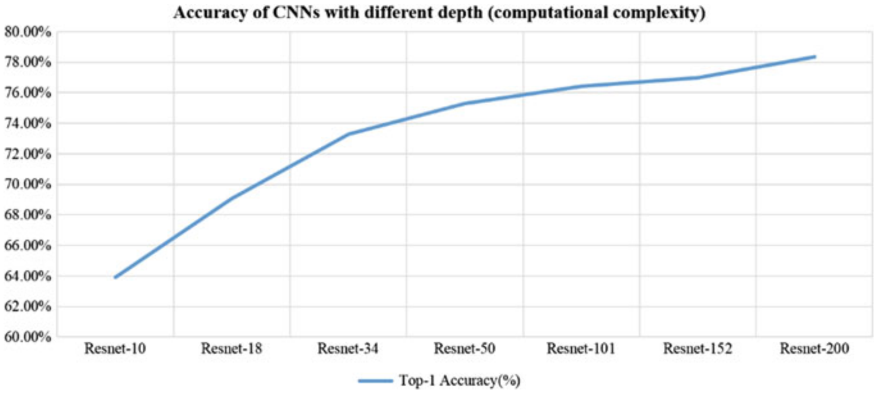


Fig. 1 The classification accuracy of CNNs with different capacity. The CNNs are with the same architecture but differ in depth. The accuracy was reported in [20–23]. The training and test images are firstly resized to 256 × N (N × 256) with the shortest edge equal to 256 and then cropped to 224 × 224. ResNet-i denotes a residual network with i layers

Figure 1 shows that the increase in capacity contributes to the improvements in accuracy on the ImageNet Dataset which is extremely large. However, the number of parameters grows exponentially as capacity and depth increase. As a result, the process of training is confronted with several prominent challenges [24]. Typical problems occur when solving the high-dimensional non-convex optimization problem are shown below:

(1) Hessian matrices suffer from ill conditions in which gradients get stuck and training slows down even in the presence of a strong gradient.
(2) There exists a large amount of local minima.
(3) The landscapes of loss functions have many high-cost saddle points which may slow down convergence.
(4) The cliffs in loss landscapes lead to exploding gradients.
(5) The improvement in accuracy often comes at the cost of computational resources.

Moreover, deep models are vulnerable to adversarial examples which are the slightly modified versions of training data [25]. As a result, it is better to apply simpler models to avoid the above-mentioned problems, especially on tasks which are not as complex as ImageNet challenge. Figure 2 shows an example. In this case, a smaller model whose potential is fully explored can perform as well as or even better than larger models on the segmentation task. The metric for evaluation is mIOU (%) [26] which divides the number of true positive pixels by the sum of numbers of true positive ones, false positive ones and false negative ones:

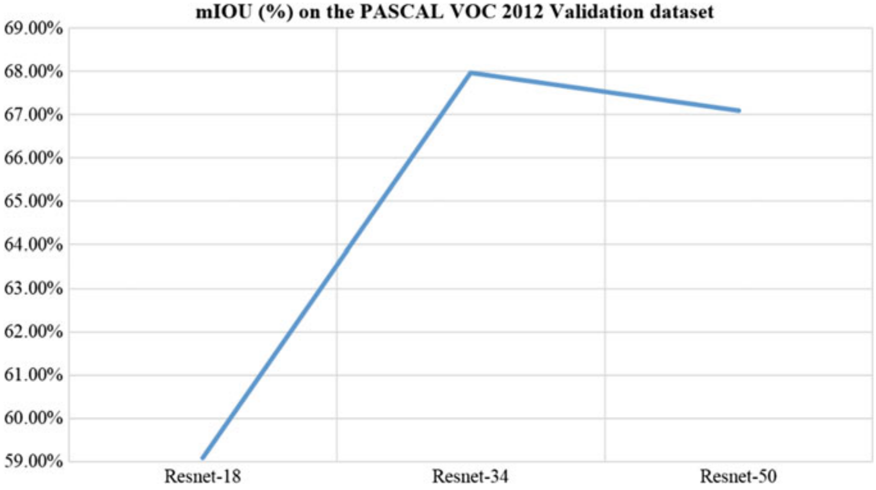


Fig. 2 mIOU (%) of segmentation models with different backbones on the Pascal VOC 2012 validation dataset for segmentation. The task-specific layers are the same for the three models. The performance was evaluated on the validation set. The metric was obtained using the development kit provided by [26]. ResNet-i denotes a residual backbone with i layers

$$mIOU = \frac{1}{N}\sum_{i=1}^{N}\frac{n_{ii}}{t_i + \sum_{j\neq i} n_{ji}} \tag{1}$$

where n_{ji} is the number of pixels of class j which are predicted to class i, and $t_j = \sum_i n_{ji}$ ($t_i = \sum_j n_{ij}$) is the total number of pixels belonging to class $j(i)$.

The Pascal VOC 2012 Dataset for segmentation contains 1,464 training images with 3,507 objects and 1,449 images with 3,422 objects for validation. The number of classes is 21. The number of images and classes are far less than those in the ImageNet classification task which has over 1,000,000 images and 1,000 classes. As can be seen from Fig. 2, ResNet-34 with 34 layers outperforms ResNet-50 with 50 layers in this simpler task. In our work, methods will be proposed to make simple networks perform as well as complex ones.

Techniques to better explore the potential of CNNs. Existing work on improving the performance of deep learning models while maintaining computational complexity can be divided into two categories. The first type focus on improving the strategies of training. Typical work of this type involves [27, 28], the former explores proper values of gradients to help convergence while the latter proposed to train CNNs layer-by-layer. In our work a 2-step scheme for optimization is proposed to train CNNs in a layer-by-layer fashion. It will be shown in Sect. 2 that the scheme outperforms the scheme proposed by [28] in the task of segmentation. The second type of work involves all types of data augmentation. In our work the gap between training data and test data in the task of segmentation is studied and the methods based on transductive learning [29] are proposed in Sect. 2.1.2 to bridge the gap between training data and test data.

Estimation of task complexity. The task of semantic image segmentation is studied in our work, it has long been a challenging computer vision task due to a lack of ground-truth and the existence of multiple types of variances or interferences. Available datasets include Look into Person [30], Multi-human Parsing [31], Microsoft COCO [32], Vistas [33], ADE20 k [34], and Cityscapes [35]. Existing research works on segmentation have evolved from R-CNN [36] and selective search [37] which conducted segmentation based on detection. A typical two-stage framework for human parsing is discussed in [38]. The two-stage methods suffer from a disadvantage that segmentation is sure to fail if the first stage provides wrong bounding boxes or saliency masks. Later work such as FCN [39] provided end-to-end schemes in which localization and pixel-level refinement can be conducted together. FCN was trained end-to-end and performed well on PASCAL VOC 2011 [40]. Another typical end-to-end architecture is the encoder-decoder structure [41]. Existing end-to-end frameworks for panoptic segmentation include [42, 43] which are also typical examples of multi-task learning.

However, nearly all of the above-mentioned models are limited in application because they are too complex to be matched to simple tasks. For instance, FCNs improve on the original version through introducing a Recurrent Neural Network (RNN) [44] but at the cost of efficiency. U-Net [41] is over-fitted to the medical image segmentation task and cannot perform as well as other models in human parsing. The structure of the decoder in DeepSaliency [45] can only be applied to a limited number of tasks. The models for semantic part segmentation [46, 47] also suffered from the lack in training data. The algorithm proposed in [46] can only be applied to the PASCAL Person Part dataset [48] and EdgeNet [49] requires the training data to have both part segment labels and boundary annotations. Moreover, Deeplab-V2 proposed in [50] and Deeplab-V3 proposed in [51] suffer from a lack of sufficient training data. The methods in [42, 43] suffer from a lack of sufficient data for panoptic segmentation. Reference [52] proposed to train only on the regions with reliable labels provided by weak supervision and image priors. The reason behind over-fitting and under-fitting in the above-mentioned tasks is the lack of analyses on task complexity and CNNs' capacity. None of the above-mentioned methods have ever considered matching the complexity of a model to that of a task.

Existing approaches to tackle over-fitting includes data augmentation and weakly supervised methods, such as BoxSup [53] and Segmenting Weakly Supervised Images [54]. Three types of weak supervision have been proposed in [29]. Incomplete supervision refers to the case where labeled data only occupies a small proportion of training data. Semi-supervised methods based on incomplete supervision are applied to exploit unlabeled data and can improve performance without human intervention. Inexact supervision refers to the cases where some supervision information is provided, but not as exact as desired. The third type of supervision is inaccurate supervision which concerns the situation where supervision is not always correct or is influenced by noises. The above-mentioned techniques have only been applied to subjectively enrich training data. The strategies favor larger models instead of smaller ones which might suffer from under-fitting. Moreover, it has not yet been evaluated whether the added data can really provide useful information which helps the model to develop more generalizable feature representations. The problems indicate the necessity of evaluating the complexity of training data to quantitatively show how complex a dataset is and how complex a model is required to be competent for the dataset (task).

Another work that have indirectly addressed the complexity of tasks is Taskonomy [55]. It has explored the relationships among visual tasks and has built a structure among multiple tasks. The redundancy across tasks was also discussed. A conclusion is drawn that through re-using the supervision from related tasks, the total number of labeled instances required for solving tasks can be reduced by 67% (compared to training one model on each task independently) with performance almost unchanged. The major objective of Taskonomy is to find the tightest set of data that is necessary for a task and remove the 67% redundancy in data. Similarly, unsupervised learning is concerned with the redundancies in the input domain and leverages the analysis to form compact representations [56]. The major contribution of Taskonomy and unsupervised learning can be concluded in two aspects. Firstly, for a dataset with a fixed size, it complexity grows when its instances are less similar to each other. Our work will also address this issue by using the number of variances to measure how complex a dataset is. The complexity evaluates how distinguished instances are. Secondly, if there exists huge redundancy in a dataset, training data can be reduced without influencing the dataset's complexity as well the performance of models trained on it. In our work, this type of dataset has a low complexity and a simpler model can perform well on it.

Three methods are proposed in Sect. 3 to evaluate the complexity of a dataset which reflects the lowest capacity required for a model to perform well on a dataset. The methods are based on the number of variances in scales, locations and the consistency between predictions from different models.

Optimization of model capacity. The definition of optimal capacity was given in [57]. The optimal capacity corresponds to the boundary between under-fitting regime and over-fitting regime. Existing research on capacity adjustment can be divided into three categories. The first category focuses on evaluating the contributions from height, width and the order of computations in a CNN to expressive power [58, 59]. Developmental learning [60] tried to enable a dataset to be enlarged continuously until the capacity of a CNN is not expressive enough. The second category of work tried to compute the information-theoretic quantities inside a CNN [61, 62]. The third type of methods focus on evaluating the model capacity with mathematical methods. Typical works are Algebraic Topology [63] and linear threshold functions [4].

The implementation of methods in the first category includes increasing either the width or height of a CNN and fine-tuning the entire network. It is well-known that deeper neurons allow for new compositions of existing neurons while wider neurons allow for the discovery of additional task-specific clues [58]. However, the increase in capacity is at the cost of efficiency. Reference [59] proposed to introduce computations of higher orders to better utilize existing features without introducing extra parameters. High-order functions enrich the hypothesis space and improve the performance of existing network models such as ResNet [11] and WRN [64] on CIFAR10 and CIFAR100 [65]. However, over-fitting is easier to occur. The second category of work aims at describing the learning of a CNN with the Information Bottleneck Principle. The entropies and mutual information within convolutional layers are computed using heuristic statistical physics methods under the assumption that weight matrices are independent. The behaviors of entropies and mutual information throughout the process of learning are studied in [61] and a conclusion is drawn that the relationship between compression and generalization are elusive. The third type of work addressed the issue of designing the architecture of a CNN by explaining how the architecture is determined by the topological complexity of a dataset. Empirical characterization of the topological capacity of a neural network model was provided in [63]. The topological phase transitions in neural networks upon the increase in datasets' complexity were introduced. Moreover, [4] provided ways of estimating the capacity of typical neuronal models, including linear and polynomial threshold gates, linear and polynomial threshold gates with constrained weights and ReLU neurons. However, only fully-connected neural networks were discussed in [4, 63]. Additionally, only very simple networks have been studied and none of the three categories of work has focused on matching the capacity of a CNN to the complexity of a high-level vision task such as segmentation. In our work, the analysis of capacity on complex models and high-level vision tasks is included. The details are given in Sect. 4.

2 Techniques to Fully Explore the Potential of Low-Capacity Networks

2.1 *Methodology*

2.1.1 Training Strategies

1. Layer-wise re-training

In this part, we propose a layer-wise training scheme which outperforms existing end-to-end training schemes as well as existing layer-wise training schemes which are proposed in [28].

It is common practice to train a neural network in an end-to-end fashion on the ImageNet dataset and then fine-tune on segmentation tasks. However, training stage-by-stage can boost the performance of a CNN model. Suppose a CNN is divided into two parts: a feature extractor and task-specific layers. We propose to train the feature extractor layer-by-layer (Fig. 3).

Denote $\mathbf{x}_K$ as the output of the feature extractor and $\mathbf{z}_K$ the prediction from task-specific layers. K is the number of layers in the feature extractor. In Training Step 1, the original feature extractor and the task-specific layers are trained together. In each step that follows, an additional layer is added to the feature extractor and the task-specific layers which have a fixed structure are re-built on top of the newly added layers. H layers are added one-by-one. The outputs of the feature extractor and those of task-specific layers are denoted as $\mathbf{x}_{K+h}$ and $\mathbf{z}_{K+h}$, $h = 1, \ldots, H$.

Denote the dataset for training as $\left\{\mathbf{x}_0^n, \mathbf{y}^n\right\}, n = 1, \ldots, N$ where N is the number of training samples. $\theta_h, h = 0, \ldots, H$ denotes the parameters of the feature extractor after the h-th layer is added, $\gamma_h, h = 0, \ldots, H$ denotes the parameters of the task-specific layers upon adding the h-th layer and fine-tuning the overall network. θ_0 and γ_0 correspond to the original network. The training process in the h-th step ($h \geq 1$) can be formalized as the minimization of the soft-max loss:

$$\left(\theta_h^*, \gamma_h^*\right) = \arg\min_{\theta_h, \gamma_h} \frac{1}{N} \sum_n L_{softmax}\left(\mathbf{z}_{K+h}\left(\mathbf{x}_0^n; \theta_h, \gamma_h\right), \mathbf{y}^n\right) \tag{2}$$

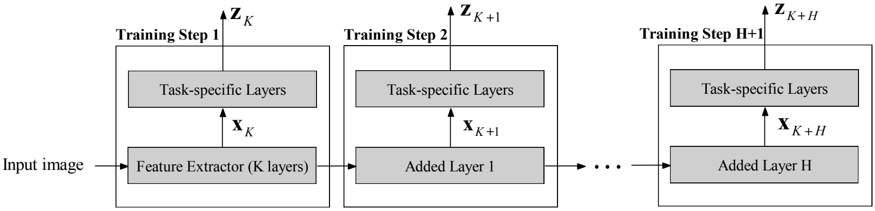


Fig. 3 The proposed scheme for adding layers to a CNN and re-training

The process of training is shown in Algorithm 1.

Algorithm 1: *Layer-wise Training Scheme*

```
1  Initialize $\theta_0$ and $\gamma_0$. Train the original network
       $\left(\theta_0^*,\gamma_0^*\right) = \arg\min_{\theta_0,\gamma_0} \frac{1}{N}\sum_n L_{softmax}\left(\mathbf{z}_K\left(\mathbf{x}_0^n;\theta_0,\gamma_0\right),\mathbf{y}^n\right)$
2  (Loop) Add Layer $h$ to the original feature extractor and randomly initialize it ($h \geq 1$).
3      (Step 1 in optimization) Optimize only the parameters in Layer $h$:
       $\left(\theta_h - \theta_h \cap \theta_{h-1}\right)^* = \arg\min_{(\theta_h-\theta_h\cap\theta_{h-1})} \frac{1}{N}\sum_n L_{softmax}\left(\mathbf{z}_{K+h}\left(\mathbf{x}_0^n;\left(\theta_h - \theta_h \cap \theta_{h-1}\right),\gamma_{h-1}\right),\mathbf{y}^n\right)$
       The layers besides Layer $h$ in the feature extractor and the task-specific layers are kept fixed.
4      (Step 2 in optimization) Optimize all the parameters in the CNN:
       $\left(\theta_h^*,\gamma_h^*\right) = \arg\min_{\theta_h,\gamma_h} \frac{1}{N}\sum_n L_{softmax}\left(\mathbf{z}_{K+h}\left(\mathbf{x}_0^n;\theta_h,\gamma_h\right),\mathbf{y}^n\right)$
5      If $h < H$, $h \leftarrow h+1$ and return to Step 2.
```

As is shown in Algorithm 1, the network is re-trained for two times upon the addition of each convolutional layer (Step 3 and Step 4). In Step 3, only the parameters in the newly added layer are optimized while in Step 4, the overall network is re-trained, $\theta_h - \theta_h \cap \theta_{h-1}$ denotes the parameters in Layer h. It is addressed in [28] that re-training upon the addition of each layer contributes to the improvements in accuracy and out-performs end-to-end training:

$$\hat{R}\left(\mathbf{z}_{K+h};\left(\theta_{h-1}^*,(\theta_h - \theta_h \cap \theta_{h-1})^*\right),\gamma_{h-1}^*\right) \leq \hat{R}\left(\mathbf{z}_{K+h-1};\theta_{h-1}^*,\gamma_{h-1}^*\right) \tag{3}$$

where $\hat{R}\left(\mathbf{z}_{K+h};\theta_h^*,\gamma_h^*\right) = \frac{1}{N}\sum\limits_n L_{softmax}\left(\mathbf{z}_{K+h}\left(\mathbf{x}_0^n;\theta_h^*,\gamma_h^*\right),\mathbf{y}^n\right)$ and $h \geq 1$. Different from [28] where only Step 3 is conducted for each added layer, all the layers in the CNN are optimized again after optimizing the single added layer:

$$\hat{R}\left(\mathbf{z}_{K+h};\theta_h^*,\gamma_h^*\right) \leq \hat{R}\left(\mathbf{z}_{K+h};\left(\theta_{h-1}^*,(\theta_h - \theta_h \cap \theta_{h-1})^*\right),\gamma_{h-1}^*\right) \tag{4}$$

$$\hat{R}\left(\mathbf{z}_{K+h};\theta_h^*,\gamma_h^*\right) \leq \hat{R}\left(\mathbf{z}_{K+h-1};\theta_{h-1}^*,\gamma_{h-1}^*\right) \tag{5}$$

It is obvious that Stochastic Gradient Descent (SGD) can at least keep the loss from increasing (4). From (4) and (5) it can be inferred that by optimizing the complete CNN after the optimization of the added layer in each step, the loss can be lower than only optimizing the added layer.

It is already shown in [28] that by adding one layer and train the added layer in each step, a network with 11 convolutional layers performs as well as VGG-13

[9] with 13 layers and the same width but trained end-to-end. Moreover, the layer-wise trained network outperforms VGG-11 [9] with 11 layers and the same width but trained end-to-end. As a result, the two-step optimization scheme proposed in Algorithm 1 contributes to larger improvements over end-to-end training schemes than the one-step scheme proposed in [28]. A comparison was made on the Look into Person Dataset [66]. There are 30,462 images for training, 10,000 images for validation and 10,000 images for testing. Layers are added to the pre-trained baseline model with 25 layers [67]. Upon the addition of each layer, training was conducted for 40,000 iterations and batch size was set to 10. The end-to-end trained network with the same structure was trained for $40,000 \times Number\ of\ layers\ added \times 2$ times. For instance, if two layers are added to the baseline model, the network constructed with Algorithm 1 is trained for 40,000 iterations for optimizing Layer 1 only, 40,000 iterations for optimizing the overall network, 40,000 iterations for optimizing Layer 2 only and 40,000 iterations for optimizing the overall network. The end-to-end counterpart is trained at once for 160,000 iterations. The initial learning rate is 2e–4 and polynomial learning policy is adopted with its power set to 0.9. The preprocessing techniques proposed in [68] is applied for both networks. Comparisons are shown in Table 1.

From Table 1, it can be inferred that the network trained with Algorithm 1 outperforms the network with the same capacity but trained end-to-end. Moreover, the networks trained with Algorithm 1 (2-step optimization) outperforms those trained using the scheme proposed in [28] (1-step optimization for each layer). As a result, Algorithm 1 can better explore the potential of a CNN without increasing the number of parameters.

2. Teacher-student method

In vision tasks, it is addressed in [69] that a larger model is better at extracting the structures from huge datasets than smaller models. However, the knowledge acquired by a large model can be transferred to a small model, the smaller model trained in this way outperforms the same network which is trained directly on the hard ground truth labels. Reference [69] proposed the way of transferring the generalization capability

Table 1 mIOU (%) of CNNs on the look into person dataset [66]

Layer-wise trained (2-step optimization proposed in Algorithm 1)	mIOU (%)
$H = 0$	45.27
$H = 1$	46.61
$H = 2$	47.53
$H = 3$	47.93
Layer-wise trained (1-step optimization proposed in [28])	
$H = 3$	47.26
End-to-end trained	
$H = 3$	47.12
$H = 4$	47.15

of a larger model to a smaller model, the method takes the activations from a large model, i.e. the soft-max activations, as soft targets for training a smaller model. The soft targets contain the similarity structures over training examples. For instance, a dog is more similar to a cat than a flower, the similarities are not provided by hard labels. The advantage of soft targets comes from the additional information which isn't included in hard targets.

To solve the problem that the probabilities of some classes are too small to influence the cross-entropy cost functions of smaller networks, distillation processing is proposed by [69]. The temperature T of the final soft-max functions can be raised until the values are suitable to train smaller networks:

$$q_i = \frac{\exp\left(z_i / T\right)}{\sum\limits_j \exp\left(z_j / T\right)} \tag{6}$$

q_i denotes the probability of an input image or a pixel to belong to class i while z_i denotes the corresponding activation. The higher the temperature T is, the softer the distribution over classes becomes.

The training data for distilling knowledge into a small model includes both original training images and unlabeled images. On the training set, the target activation is a weighted sum of the soft target produced by the large model and the hard target produced by the ground truth labels with temperature equal to 1. The soft target has a high temperature in the soft-max function. The weight of the former should be larger than that of the latter and the magnitude of the first term should be multiplied by T^2 to cancel the influence brought by the changes in temperature.

The teacher-student method is applied in our work, as will be addressed in Sect. 3.

2.1.2 Strategies of Data Augmentation

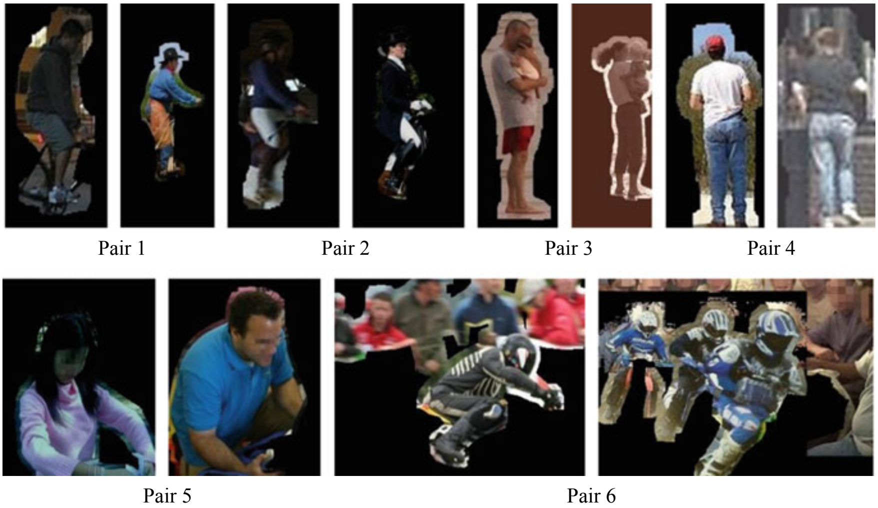
Unlike the scenarios in a classification task where the gap between training accuracy and test accuracy is lower than 3% as long as the training set and the test set have similar distributions, the gap in a segmentation task is always larger than 10%. This indicates a pixel-specific task is more sensitive to the (small) variations than an image-specific task. The variations in the image segmentation task include the changes in poses, color, illumination and so forth. In our work, two methods are proposed to bridge the gap between the training data and test data. The variances in the test data are studied to enrich the training data. With the variances of the training data being more comprehensive, the potential of a model can be better explored. As was introduced in [29], incomplete supervision concerns the situation in which a small amount of labeled data and abundant unlabeled data are available. In that case, the labeled data is insufficient to train a good learner. The unlabeled data are exploited by semi-supervised learning in addition to labeled data to improve learning performance.

Transductive learning proposed by [29] is a special type of semi-supervised learning and assumes that the available unlabeled data is exactly the test data. The two methods of data augmentation proposed in our work are similar to transductive learning by exploiting unlabeled test data to improve performance.

1. Bridge the gap between poses

The first method for data augmentation is through generating training images which have similar poses to those from the test set. Besides a CNN model for segmentation, a model for human pose analysis was trained, as is introduced in our previous work [68]. Skeleton detection is a simpler task than person part segmentation and there are more training data samples available. As a result, the predictions from the model for pose analysis tend to be more generalizable. In the proposed method, each image is firstly divided into regions each of which contains one person. For each region, a vector describing poses is predicted. An algorithm is developed to find the person from training data with the most similar gesture to each person in test data. Upon matching each pair of people, homography transformation is conducted on the people from training images to make them more similar to those in test images. The ground truth masks for training images are transformed in the same way. The transformations produce mocked test images which are similar to test images but with labels. Some examples are shown in Fig. 4. People have been segmented out for clear demonstration.

As can be seen from Fig. 4, a mocked test set with labels is generated based on training data. The transformations are based on the coordinates of predicted joints.


Pair 1

Pair 2

Pair 3

Pair 4

Pair 5

Pair 6

Fig. 4 Examples showing six pairs of images containing people with similar poses. Within each pair, the left image is from the test set and the right one is from the training set. We have searched the training set to find the person with the most similar pose to each person in each test image. Homography transformations is conducted on the foreground regions from training images

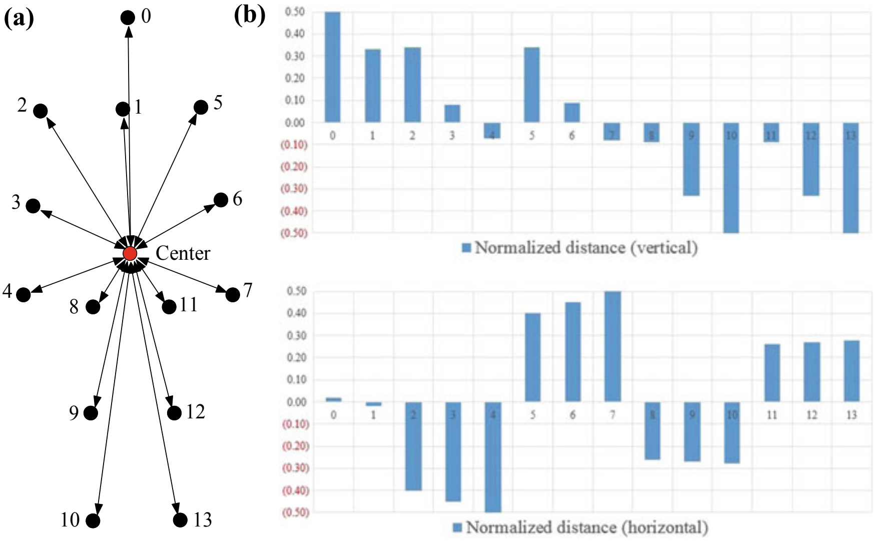


Fig. 5 The proposed descriptor for describing poses. **a** The indices of joints. **b** The offsets of different joints from the center. The offsets are normalized with respect to the maximal distance between different pairs of joints. The top histogram describes the normalized offsets in the vertical direction while the bottom one describes those in the horizontal direction

The mocked test set can enrich the training set and bridge the gap between training data and test data. Experiments will show that the involvement of the mocked test set during training improves the performance of segmentation.

To evaluate the similarity in poses, a feature descriptor is developed in our work. Firstly the definition of 14 joints is based on the discussion in [70]. The 14 joints are nose, neck, left shoulder, left elbow, left wrist, right shoulder, right elbow, right wrist, left hip, left knee, left ankle, right hip, right knee and right ankle. A center point can be computed by averaging the coordinates of the predicted joints, it is shown by the red dot in Fig. 5a.

Figure 5 describes the proposed descriptor, two histograms are computed to describe the pose of each person in the form of horizontal and vertical translations. The similarity in pose is evaluated by the sum of Euclidean distances between the two pairs of histogram vectors. For a test image with multiple people, its mocked counterpart is composed of foreground regions from different training images, an example is shown in Pair 6 in Fig. 4. The existence of other types of variances, such as rotations, scaling or changes in illumination may still cause the gap between training data and test data. As a result, the data augmentation in color variances is still required.

2. Bridge the gap in color

Besides the variances in pose, the differences in color also contribute to the gap between training set and test set. In our work, a self-supervised CNN for colorization is adopted to learn on the test set, the model is used to colorize the grayscale versions of training images based on the distributions of colors which are learned from the test set. It is already shown in Fig. 4 that the training images can be used to generate mocked test images. Similarly, with the joint usage of the colorization model, the mocked test images can show more similarity in color to real test images.

Similar to human parsing, colorization is also a pixel-specific task. As a result, our model for segmentation can be used to conduct colorization by making changes to the loss function. According to the method proposed in [71], an RGB image is firstly converted to Lab format. The L-channel corresponds to light while a-channel and b-channel correspond to colors. As a result, L-channel can be treated as inputs and the remaining two channels as ground truth outputs. Suppose the height and width of an input image are H and W, respectively. Denote $\mathbf{X}^{H\times W\times 1}$ as an input image and $\mathbf{Y}^{H\times W\times 2}$ the ground truth. As is addressed in [71], colorization can be divided into two steps. In the first step, a distribution over ab-values $\hat{\mathbf{Z}} \in [0, 1]^{H\times W\times Q}$ is predicted for each pixel where Q is the number of possible ab pairs, the distributions are converted to color labels in the second step:

$$L\left(\hat{\mathbf{Z}}, \mathbf{Z}\right) = -\sum_{h,w} weight(h, w) \sum_{q} \mathbf{Z}_{h,w,q} \log\left(\hat{\mathbf{Z}}_{h,w,q}\right) \tag{7}$$

where $weight(h, w)$ denotes the weights on pixels, the most frequently appeared pixels are assigned lower weights. All pixels are assumed to subject to an independent and identical distribution which is the sum of the color distribution from ImageNet training set $\mathbf{p}$ and a uniform distribution $\frac{1}{Q}$:

$$weight(h, w) \propto \left(0.5\mathbf{p} + \frac{0.5}{Q}\right)^{-1} \tag{8}$$

In the second step, the predicted distribution mask $\hat{\mathbf{Z}} \in [0, 1]^{H\times W\times Q}$ is converted to color values using a soft-max function with temperature equal to 0.38 for each pixel.

The two parts mentioned above can produce mocked test images which are similar both in colors and poses to real test images. The involvement of the generated images during training can improve the performance of human parsing, as is shown in Table 2.

Table 2 Improvements on mIOU (%) brought by data augmentation on poses and colors

Method	mIOU (%)
Original Deep Lab-V2 [50]	64.94
Deep Lab-V2 [50] with data augmentation applied	65.41
Original segmentation module proposed in [68]	67.43
Segmentation module proposed in [68] with data augmentation applied	67.94

From Table 2 it can be inferred that the proposed two ways of bridging the gap between training data and test data can improve the performance of different models without increasing computational complexity. In other words, the potential of CNNs are better explored.

3 Estimation of Task Complexity

3.1 Methodology

3.1.1 Methods Based on the Variances in Scales

Similar to detection tasks, the performance of segmentation is influenced by the variances in spatial dimensions and translations. For instance, it is shown in [72] that prior information about the spatial dimensions of anchor boxes is important in improving average IOU. Spatial dimensions include scales and aspect ratios. Reference [72] showed that by selecting appropriate prior anchors that fit well to the sizes of objects in the interested dataset, the average IOU could be improved by 2.3% than applying the same model without prior information. Moreover, [72] showed that the more priors there are and the more accurate the priors are, the higher performance becomes. That is to say, even if the spatial dimensions of objects can be predicted by a detection model such as [72], accurate prior is also necessary because the performance is limited by the model's capacity or expressive power. The more variances in spatial dimensions there are, the harder a task becomes.

In our work, a method is proposed to evaluate the scale variances in the task of human parsing. We use the model for pose estimation to evaluate the scales of human objects in human parsing datasets. It is well known [46] that pose estimation and human parsing are complementary. A model for pose estimation easily suffers from occlusions which can be fixed by a model for human parsing. In another way, a model for pose estimation can provide object-level feature representations and regularity to help a human parsing model in aligning with instances, especially in the case where some human parts share the same color with the background. More importantly, models for pose estimation are better at detecting objects in small scales than human parsing models.

Figure 6 shows the predictions from the model for pose estimation. The people in the three examples cannot be detected by human parsing models because of small scales. The existing state-of-the-art CNN models for human parsing, such as PSPNet [73], fails in segmenting out people from the third row in Fig. 6. However, the model

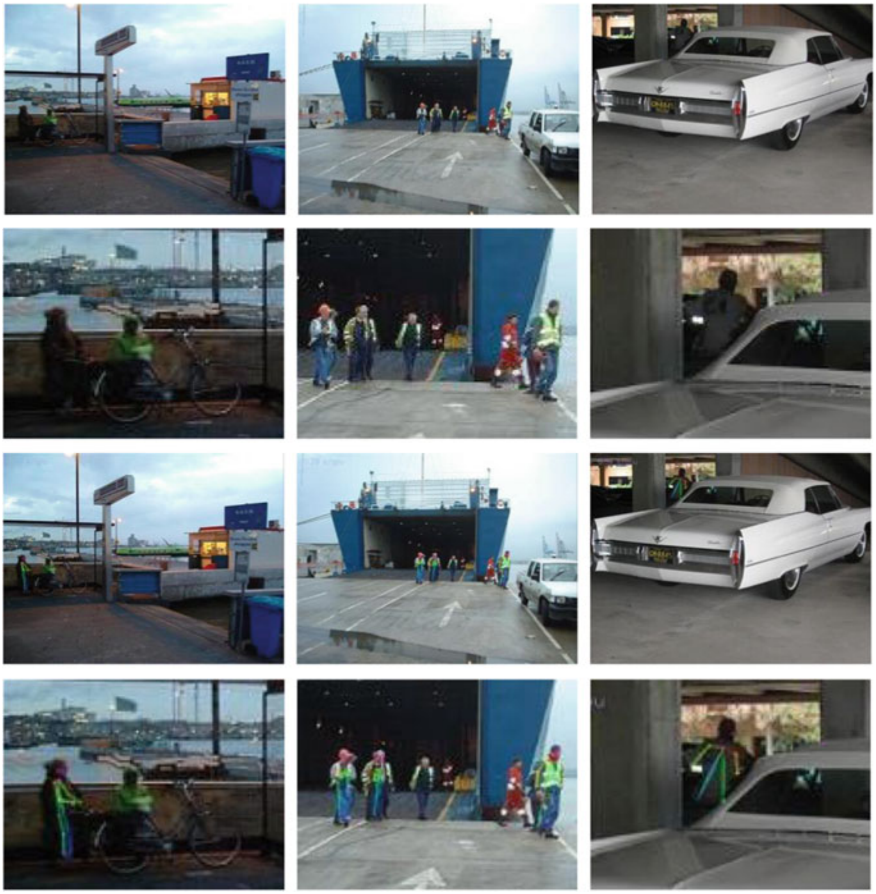


Fig. 6 Images with small targets. The first and second rows: the input images. The third and fourth rows: the results of pose estimation. The second and fourth rows are cropped from the first and third rows for clear demonstration. The images come from the Pascal VOC 2010 dataset and the skeletons were predicted by the method in [70]

for pose estimation [74] with much less parameters than [73] can localize the people from images. As a result, the model for pose estimation is more robust to the variances in scales and we adopt such a model to detect people and evaluate their scales based on Algorithm 2 (Fig. 7).

Algorithm 2: *Pose estimation on instances of different scales.*

```
Input:   Images with people in all scales.
Output:  Predicted skeletons describing poses.
1        Directly apply the model [74] on each input image.
2        Divide the results from Step 1 into 3 categories:
           1. The predicted skeletons occupy a region that is large enough in the image (The
              width of the predicted region is no less than one fifth the width of the image. The
              height of the predicted region is no less than a half of the image).
           2. The area of the predicted region is smaller than the threshold which is set in the
              above category but greater than 0. In this case, crop a rectangle region from the
              input image and enlarge it. The cropped region centers at the centroid of the
              detected region.
           3. If nothing is detected. Then crop five regions from the input image according to
              the strategy in Fig. 7 (b) and enlarge the regions. The five regions are equally
              sized. If nothing can be detected from the cropped regions. Iteratively conduct
              the strategy iteratively until targets are successfully detected.
3        Assign a tight bounding box for each detected person and record the relative size of
         each box with respected to the width and height of the input image.
```

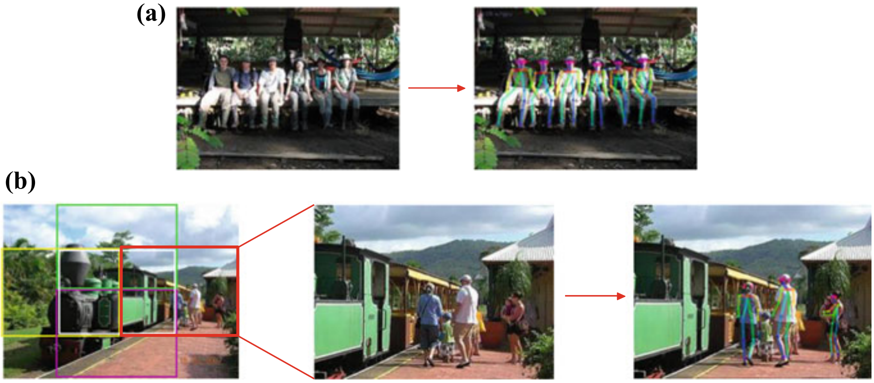


Fig. 7 The strategy of pose estimation on small targets. **a** Pose estimation for targets in a normal size. It is a one-step operation. **b** For small targets, detection is likely to fail in the Step 1 in Algorithm 2. In that case, five regions which are most probable to contain targets are cropped and enlarged. In this example, the five cropped regions are highlighted in green, yellow, white, red and purple. Detection in the red region is successful in Step 2 in Algorithm 2. If detection still fails, iterative cropping and detection are conducted

According to Algorithm 2, the proportions that targets take up in images can be computed. We also conduct analysis on the distribution of the proportions. The more even the distribution is, the more variances there are in the scales of targets, and the harder the task becomes. Figure 8 shows the distribution of proportions that objects take up in images. The horizontal axis denotes the proportions of area in images that are taken up by targets, the vertical axis denotes the percentage of images that satisfy the conditions. From Fig. 8a it can be seen that training images include objects of all scales.

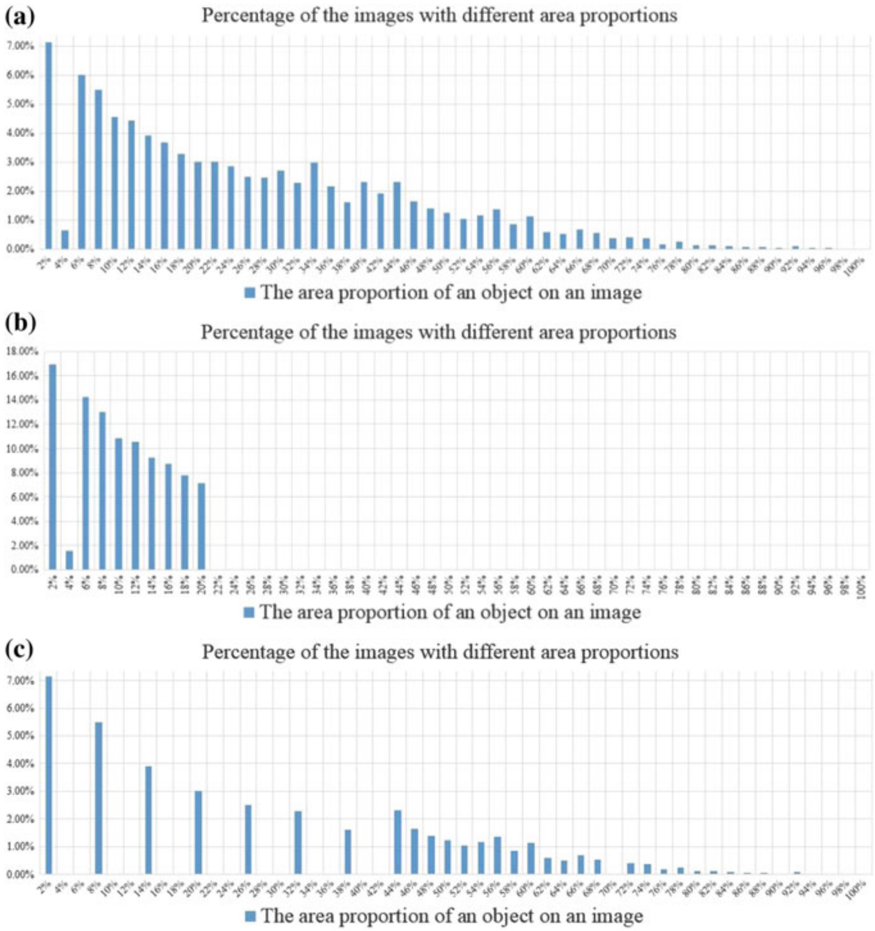


Fig. 8 Percentages of objects that take up different proportions of areas in images. **a** The overall dataset for training and test [48] with 3,533 images. **b** 1,489 images are sampled from the dataset, the images are with similar scales. **c** Another 1,489 images with greater divergence in scales are sampled

Table 3 Comparison on the difficulties of subsets

Subset for training and test	mIOU (%)
Subset 1 shown in Fig. 8b	59.47
Subset 2 shown in Fig. 8c	43.26

In can be seen that the images in the subset in Fig. 8b include the objects with similar scales and those in the subset in Fig. 8c contain the objects of significantly different scales. We conducted experiments on the two subsets. For both subsets, a random division was performed with 70% for training and 30% for testing. The same segmentation model based on VGG-16 was used and initialized in the same way. Batch size was set to 10 and 20,000 iterations were conducted for training. It is shown in Table 3 that the subset (b) is easier for the model to fit well. As a result, scale variance is an important factor influencing the complexity of a segmentation task.

3.1.2 Methods Based on the Variances in Locations

The evaluation using the variance in locations is done in the similar way. The distribution of translations of objects in images is computed. The more even the distribution is, the harder a task becomes. A task becomes easier when objects in all images are of similar translations.

3.1.3 Methods Based on the Consistency of Predictions from Different Models

We also propose a third method for evaluating the complexity of a dataset: using the consistency in the predictions from different models. In our case, the consistency between a human parsing model and a model for pose estimation is evaluated. Both the model for pose estimation and that for human parsing focus on predicting the profiles of persons. As a result, we can generate the profile of a person by conducting dilation on the predicted skeletons. For a task that is simple enough, both models can perform well and thus the profiles predicted by them should be similar. If a task becomes difficult and contains small objects or occlusions, one of the models may fail and cause the discrepancy between their predictions. For a task that is difficult enough, both models may fail in different ways and it is more likely that their predictions are dissimilar. As a result, the similarity in the predictions from both models can help us judge the difficulty of a task.

3.2 Summary

This section proposes three methods to evaluate the difficulty of a segmentation task. This is especially useful when a practical dataset is not as complex as the training dataset. In that case, we can use the similarity in scales, locations or the consistency in predictions to choose a subset from training data whose complexity matches that of a practical scenario well. A smaller model can be trained to perform well enough in the practical scenario. More importantly, our work provides a way to justify the validity of techniques for data augmentation. If the complexity of a dataset does not increase after data augmentation, the data augmentation is useless.

4 Optimization of Model Capacity

4.1 Methodology

4.1.1 Capacity Optimization by Improving the Orthogonality Between Kernels in a CNN

In a CNN, different kernels within the same layer extract complementary cues that address the target task. As was discussed in [61], the training of a CNN can be divided into a fitting phase and a compression phase. In the compression phase, the mutual information between hidden representations and inputs drops while the mutual information between hidden representations and ground truth labels increases. The representations with less features about inputs generalize better. As a result, the mutual information between hidden representations and labels is a decisive factor in determining the performance of a CNN model.

It is sure that for one convolutional layer, the more independent kernels are, the less channels are required to obtain a fixed amount of information. As a result, independent kernels can compose a more compact but as efficient feature representation compared with correlated kernels. Less parameters are required and its capacity can be better matched to a task. Besides the optimization on capacity, the orthogonality between kernels also shows an important influence on the training process as well as the performance of a CNN. This has been discussed in [3].

For a CNN with L layers, suppose the number of input *channels* to the $l-th(1 \leq l \leq L)$ layer is d and the number of output channels is n. $\mathbf{x}$ denotes the input to the layer and $\mathbf{h} = \mathbf{W}_l\mathbf{x}$ where $\mathbf{W}_l$ is the weight matrix, $\mathbf{W}_l \in \mathbb{R}^{n \times d}$ and $\mathbf{W}_l\mathbf{W}_l^T = \mathbf{I}^{n \times n}$. As is usually the case, we substract inputs by their mean values and normalize the results to be within $[-1, 1]$. $\mathbf{x}$ has the properties of zero means and unit covariances: $E_{\mathbf{x}}[\mathbf{x}]$=0 and $cov(\mathbf{x}) = \sigma^2\mathbf{I}$. Then

$$E_{\mathbf{h}}[\mathbf{h}] = \mathbf{W}_l^T E_{\mathbf{x}}[\mathbf{x}]=0 \tag{9}$$

The covariance of **h** can be computed as

$$\begin{aligned} cov(\mathbf{h}) &= E_{\mathbf{h}}[\mathbf{h} - E_{\mathbf{h}}[\mathbf{h}]]^2 \\ &= E_{\mathbf{x}}[\mathbf{W}_l(\mathbf{x} - E_{\mathbf{x}}[\mathbf{x}])]^2 \\ &= E_{\mathbf{x}}[\mathbf{W}_l(\mathbf{x} - E_{\mathbf{x}}[\mathbf{x}])] \cdot E_{\mathbf{x}}[\mathbf{W}_l(\mathbf{x} - E_{\mathbf{x}}[\mathbf{x}])]^T \\ &= \mathbf{W}_l E_{\mathbf{x}}[(\mathbf{x} - E_{\mathbf{x}}[\mathbf{x}])] \cdot E_{\mathbf{x}}[(\mathbf{x} - E_{\mathbf{x}}[\mathbf{x}])]\mathbf{W}_l^T \\ &= \mathbf{W}_l cov(\mathbf{x})\mathbf{W}_l^T \\ &= \sigma^2 \mathbf{W}_l \mathbf{W}_l^T \\ &= \sigma^2 \end{aligned} \tag{10}$$

For networks with ReLU non-linearities, (9) can be achieved by Batch Normalization layers and Scale layers. For other types of non-linearities such as tanh, (9) and (10) can be directly achieved. As a result, orthogonal weight matrices can maintain the properties of normalization and de-correlation between channels, according to [75]. As is discussed in [76], the normalized and de-correlated activations contribute to the improvement on the conditioning of the Fisher information matrix of the weights in each layer, and the better conditioning accelerates the learning process of neural networks. For the l-th$(1 \leq l \leq L)$ layer with weight matrix $\mathbf{W}_l$ and input **x**, the Fisher information matrix is denoted as:

$$F_{\mathbf{W}_l} = E_{\mathbf{x}\sim p(\mathbf{x})}\left\{E_{\mathbf{y}\sim p(\mathbf{y}|\mathbf{x},\mathbf{W}_l)}\left[\left(\frac{\partial \log p(\mathbf{y}|\mathbf{x},\mathbf{W}_l)}{\partial \mathbf{W}_l}\right)\left(\frac{\partial \log p(\mathbf{y}|\mathbf{x},\mathbf{W}_l)}{\partial \mathbf{W}_l}\right)^T\right]\right\} \tag{11}$$

For ease of computation, we discretize both the input **x** and the output from the nonlinear function **y** into bins. In the case of binary classification, the possible values of **y** lie in two bins. $p(\mathbf{y} = 1|\mathbf{x}, \mathbf{W}_l) = f_l(\mathbf{W}_l\mathbf{h}_{l-1} + \mathbf{b}_l)$ and $f_l()$ is the nonlinear function. Equation (11) can be converted to

$$F_{\mathbf{W}_l} = E_{\mathbf{x}\sim p(\mathbf{x})}\left\{E_{\mathbf{y}\sim p(\mathbf{y}|\mathbf{x},\mathbf{W}_l)}\left[\mathbf{vec}\left(\boldsymbol{\delta}_l\mathbf{h}_{l-1}^T\right)\mathbf{vec}\left(\boldsymbol{\delta}_l\mathbf{h}_{l-1}^T\right)^T\right]\right\} \tag{12}$$

where $\boldsymbol{\delta}_l$ denotes the back-propagated gradient through $f_l()$ and $\mathbf{vec}(\mathbf{X})$ denotes the vector containing all the elements in matrix **X**. It is a column vector concatenating the transposed version of all rows in **X**. Based on the independence between $\boldsymbol{\delta}_l$ and $\mathbf{h}_{l-1}$, (12) can be converted to

$$F_{\mathbf{W}_l}(km, ln) = E_{\mathbf{x}\sim p(\mathbf{x})}\left\{E_{\mathbf{y}\sim p(\mathbf{y}|\mathbf{x},\mathbf{W}_l)}[\boldsymbol{\delta}_l(k)\boldsymbol{\delta}_l(l)]\right\}E_{\mathbf{x}\sim p(\mathbf{x})}\left[\mathbf{h}_{l-1}(m)\mathbf{h}_{l-1}(n)\right] \tag{13}$$

$F_{\mathbf{W}_l}(km, ln)$ evaluates the statistical correlation between the $(k, m) - th$ entry and the $(l, n) - th$ entry in matrix $\mathbf{W}_l$. The conditioning of $F_{\mathbf{W}_l}(km, ln)$ can be improved by the following constraint:

Table 4 Influences on mIOU (%) brought by reducing channels

Method	mIOU (%)
Original segmentation model (VGG-16 based) [50]	56.53
Segmentation module with conv5_3 compressed using the method in 4.1.1	56.57
Segmentation module with conv5_3 and conv5_2 compressed using the method in 4.1.1	56.48

$$E_{\mathbf{x}\sim p(\mathbf{x})}\left[\mathbf{h}_{l-1}\mathbf{h}_{l-1}^{T}\right] = \mathbf{I} \tag{14}$$

According to (9) and (10), orthogonal weight matrices can contribute to (14) which improves the conditioning of Fisher information matrices and thus accelerates the convergence of learning. For the first term in (13), it is discussed in [76] that the possible correlations brought by nonlinear functions such as ReLU can be ignored.

Moreover, if $n = d$ for $\mathbf{W}_l \in \mathbb{R}^{n\times d}$, we have $\mathbf{W}_l^T\mathbf{W}_l = \mathbf{W}_l\mathbf{W}_l^T = \mathbf{I}$ and $\|\mathbf{h}\|{=}\|\mathbf{x}\|$. The values of gradient can also be maintained:

$$\left\|\frac{\partial L}{\partial \mathbf{x}}\right\| = \left\|\frac{\partial L}{\partial \mathbf{h}}\mathbf{W}\right\| = \frac{\partial L}{\partial \mathbf{h}}\mathbf{W}\mathbf{W}^T\frac{\partial L}{\partial \mathbf{h}}^T = \frac{\partial L}{\partial \mathbf{h}}\frac{\partial L}{\partial \mathbf{h}}^T = \left\|\frac{\partial L}{\partial \mathbf{h}}\right\| \tag{15}$$

So orthogonal filters helps to keep the stability of both the norms of activations and the gradients.

In our work, better convergence (higher accuracy as is shown in Table 4) and the reduction in the number of feature channels are achieved at the same time. Different from [3] which constructed an orthogonal set of kernels from random initialization, we pre-trained a CNN on the human parsing task and then select a most compressed set of orthogonal kernels that span almost the same space as the pre-trained kernels. The kernels in the compressed set is half as many as the original set of kernels. The method for reducing the number of kernels is based on [75]. It produces a set of orthogonal kernels and we select half of them which can best explain for the variances in the pre-trained kernels.

A set of kernels with less elements than the set in the pre-trained layer is achieved with two advantages: (1) the number of parameters is reduced with distinguished features not being reduced and (2) better orthogonality ensures better training. In our experiments, the numbers of channels in the last two layers of the feature extractor in the pre-trained model [50] are reduced by a half. Table 4 shows that performance will not be influenced by compression.

4.1.2 Capacity Optimization by Reducing the Dependency Among Kernels in CNNs

Besides the number of convolutional kernels, the dependency among pre-trained kernels is also an important factor indicating the complexity of a task. For a fixed number of convolutional kernels, the more complex the task is, the more non-linear operations should be conducted on discriminative features, and more independent kernels are required to perform well. We propose to firstly train a CNN model on a task to achieve the highest possible accuracy, then evaluate the number of independent components in the set of pre-trained kernels using principal component analysis (PCA) [77]. The kernels of a layer are regarded as random variables in a high-dimensional space. If more components are required to account for the variability in the variables, the task is more complex and vice versa.

Suppose the input to a layer has M channels and the number of output channels is N. The weights in the layer includes $M \times N$ kernels. The kernels are formed as vectors. The size of a vector is $1 \times K^2$ for a $K \times K$ kernel. N kernels are organized into a $K^2 \times N$ matrix $\mathbf{X}$. We will divide our discussion of adjusting the linear dependency between kernels into two parts. In the first part, the way of initializing the weights in the new convolutional layer is addressed. The procedure is described in Algorithm 3 and conducted once for each input channel. In the second part, the structure of the new convolutional layer with adjustable capacity will be introduced.

Algorithm 3 *Initialize the new convolutional layer with pre-trained weights*

```
1 Organize the $N$ vectors and construct a $K^2 \times N$ matrix $\mathbf{X}$.
2 Subtract each entry by the mean of that row.
3 Compute the normalized covariance matrix $\mathbf{C} = 1/N\left(\mathbf{X}\mathbf{X}^{\mathrm{T}}\right)$.
4 Conduct singular value decomposition (SVD) on $\mathbf{C}$ and record the singular values as well as singular vectors.
5 The variance ratio is the sum of $H$ largest singular values divided by the sum of all singular values. If a variance ratio is larger than 85%, we consider the representation power of the simplified kernels to be with enough expressive power.
6 Construct a $H \times K^2$ matrix $\mathbf{P}$ whose rows are the singular vectors corresponding to the $H$ largest singular values. ($H < K^2$)
7 Approximate the original $N$ vectors with the columns of $\hat{\mathbf{X}}$ where $\hat{\mathbf{X}} = \mathbf{P}^{\mathrm{T}}\mathbf{P}\mathbf{X}$.
```

In Algorithm 3, the H row vectors in matrix $\mathbf{P}$ are linearly independent. Each of the N kernels in $\mathbf{X}$ can be represented by a linear composition of the H vectors. For a fixed variance ratio, if a larger H is required, the pre-trained kernels are less dependent to perform well on the task, and thus the complexity of the dataset is higher. In the extreme case, if $H = 1$, all the N kernels in $\mathbf{X}$ are linearly correlated.

The structure of the new convolutional layer with adjustable capacity is illustrated in Fig. 9a and b. Figure 9a shows the way of decomposing one traditional layer into two sub-layers. The second layer includes 1×1 convolutions for approximating the original $M \times N$ kernels with a linear combinations of $M \times H$ principal components. Figure 9b shows how the proposed convolutional layer is different from a traditional

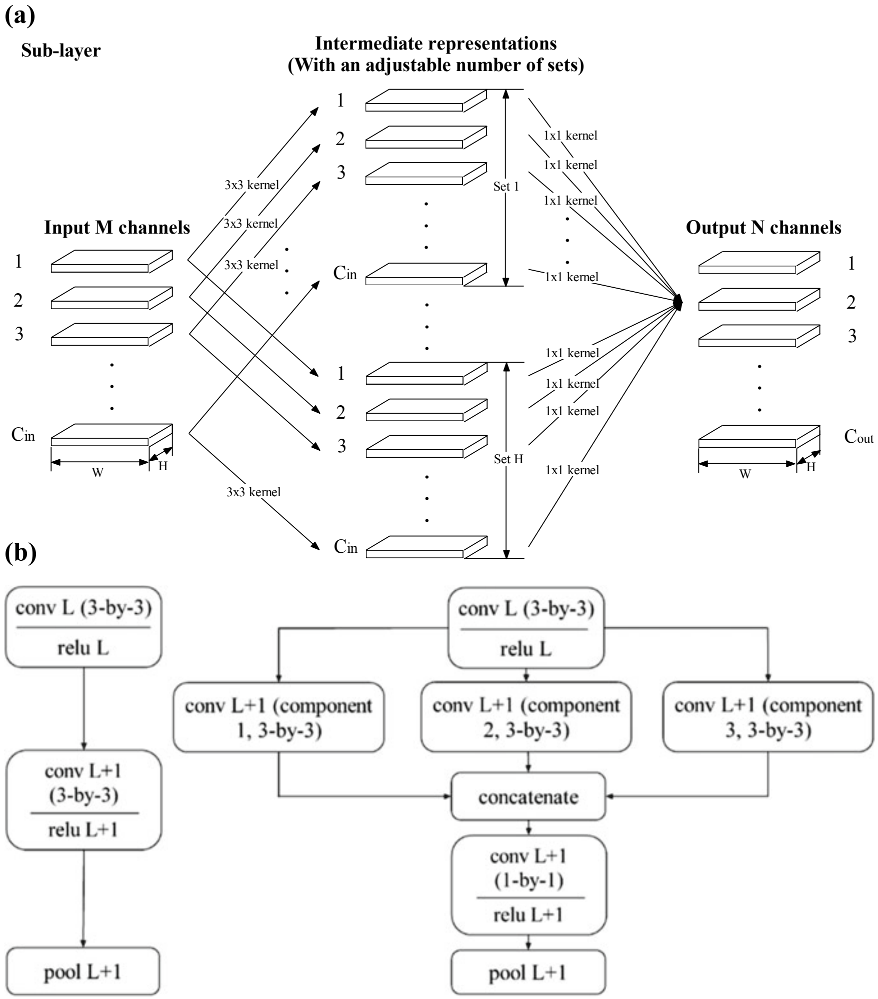


Fig. 9 The structure of the new convolutional layer with adjustable capacity. **a** One traditional layer is decomposed into two sub-layers. The first sub-layer has HC_{in} output channels while the second has C_{out} outputs. **b** The structure of a traditional convolutional layer and the proposed layer with $H = 3$ principal components

one. “conv L (3-by-3)” and “conv L+1 (3-by-3)” are conventional convolutional layers with 3-by-3 convolutional kernels. Each layer has $M \times N$ kernels. “conv L+1 (component i, 3-by-3)” $i = 1, 2, 3$ are special layers. Each layer has only M convolutional kernels each of which corresponds to one input feature channel. “conv L+1 (1-by-1)” is a conventional 1-by-1 layer with $3M \times N$ convolutional kernels.

After pre-training and applying Algorithm 3 on the kernels, the initialized new convolutional layers undergo the second round of training. In our experiments, if the following three conditions are satisfied, the capacity is better matched to the complexity of the task: (1) H components can explain over 85% of the variances in the original kernels. (2) The loss in training accuracy is less than 1% after re-training. (3) After data augmentation and re-training, the loss in test accuracy is below 1%. Note that the number of discriminative features in the simplified layer is less than that in the original layer. Moreover, for a network that is either small or large, only less than 100% of the features are generalizable. As a result, we have to make the proportion of useful features in the simplified network larger the proportion in the network before simplification, this can be achieved by data augmentation.

For $H = 3$, the reduction in the number of parameters is around 67%. If the task becomes more complex, a higher H is required. The number of principal components can be chosen from one to eight. This is due to the fact that in the PCA analysis, each input variable is a 3-by-3 convolutional kernel with 9 dimensions, and that the network with nine components has the same capacity as the network before simplification. The strategy is starting from the median number of eight, that is, four. If the simplified network with four principal components cannot satisfy the above-mentioned three conditions, then try five, six, seven, and eight components in turn; otherwise try one, two and three components in turn. The process is conducted recursively. For each number of principal components, the network is re-trained and tested. If none of the simplified networks can perform as well as the original network, then we have to use the original network for inference. The maximum number of re-training is four.

4.1.3 Capacity Optimization by Reducing the Redundancy Among Kernels in Convolutional Layers

Suppose the capacity of a CNN model is more than enough for a task, if the CNN model is trained in an improper way, or pre-trained on another dataset, redundancy may appear among convolutional kernels. That is to say, removing some kernels and related inter-connections brings no harm to performance.

A method of matching the capacity of a CNN to a task is studied by us. It is implemented by adjusting the number of uncorrelated feature channels. Our proposed model is trained on the PASCAL VOC 2010 Person Part dataset [48, 78] and the COCO dataset [32] which include 3,533 images and 80,000 images, respectively. The datasets are far smaller than the one used in ImageNet competition [79] which includes more than 1,000,000 images for training. As a result, our model should not be as complex as the feature extractor in Deeplab-V2 model [50] which is matched to the complexity of the ImageNet dataset.

It is addressed in [80] that the use of a Gaussian mixture model for modeling the distribution of weights in a neural network is appropriate. Each data point corresponds to a kernel and each Gaussian component is a collection of kernels. In a CNN, different kernels within one layer correspond to different clues for the task while different layers correspond to different compositions of clues. The clues and compositions in a CNN can be clustered based on their similarities. As a result, a CNN can be divided into functional units. The feature channels within the same functional unit share similar semantic meanings. Whether a functional unit is useful or not for the task is determined by the influence on the overall performance brought by removing the unit. We propose to implement this based on the EM algorithm [81] to reduce the number of functional units. This process can overcome the over-fitting problem of a CNN model.

Algorithm 4 is conducted once for each layer and firstly on the highest layer in the feature extractor. Suppose that the input to a certain layer has M channels and the output has N channels. The M kernels corresponding to each of the N output channel are concatenated to produce $\mathbf{x}_i, i = 1, \ldots, N$. $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. A Gaussian mixture model is constructed to cluster the N concatenated vectors into K groups. A functional unit denotes an assembling of similar vectors. $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_K\}$, $\boldsymbol{\Sigma} = \{\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \ldots, \boldsymbol{\Sigma}_K\}$ and $\boldsymbol{\pi} = \{\pi_1, \ldots, \pi_K\}$ denote the parameters of the K clusters where $\mu_i, i = 1, \ldots, K$ denote the mean vectors, $\Sigma_i, i = 1, \ldots, K$ denote the covariance vectors and $\pi_i, i = 1, \ldots, K$ denote the mixing coefficients.

Algorithm 4 *Partitioning a CNN layer into functional units*

```
1  Initialize $K, \boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_K\}, \boldsymbol{\Sigma} = \{\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, ..., \boldsymbol{\Sigma}_K\}$ and $\boldsymbol{\pi} = \{\pi_1, \mathrm{K}\ , \pi_K\}$
2  (Outer loop) Increase $K$
3  (Inner loop) Perform E step to evaluate the responsibilities
   $\gamma(z_{nk}) = \frac{\pi_k N(\mathbf{x}_n | \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j N(\mathbf{x}_n | \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)}$
4  Perform M step to update $\boldsymbol{\mu} = \{\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, ..., \boldsymbol{\mu}_K\}, \boldsymbol{\Sigma} = \{\boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, ..., \boldsymbol{\Sigma}_K\}$ and
   $\boldsymbol{\pi} = \{\pi_1, \mathrm{K}\ , \pi_K\}$
5  Check the convergence of $\ln p(\mathbf{X} | \boldsymbol{\mu}, \boldsymbol{\Sigma}, \boldsymbol{\pi})$. If the convergence criterion (reaching
   maximum) is not satisfied, return to Step 3, else proceed.
6  Evaluate the influence of halving the number of clusters within one layer on the
   performance by comparing the test accuracy before and after the process. If the drop
   in accuracy is below a threshold (3%), stop, else return to Step 2.
```

In the beginning, $K = 1$. Halving the cluster will cause a great loss in accuracy because for a large cluster, different functional units are integrated in the cluster and the operation of halving a cluster in Step 6 leads to a complete loss of some functional units. The number of clusters K increases until each cluster represents one functional unit and halving one cluster only results in a partial loss of one functional unit. There are two loops in the algorithm. In the outer loop, K increases and the number of elements in each cluster decreases. The system accuracy will not drop significantly since parts of all functional units are still kept. The threshold for evaluating the drop in accuracy is chosen to be 3%.

As is mentioned above, if the removal of a functional unit bring no harm to performance, it can be removed. Table 5 shows the influences on performance of removing each functional unit in layers conv6_1, conv6_2 and conv6_3 from the network in [50]. The network is re-trained an augmented dataset after the removal of each functional unit.

As is shown in Table 5, there is one functional unit in each layer whose absence brings no harm to test accuracy as well as training accuracy. The removal of the 5th unit in conv6_2 contributes to an improvement in test accuracy and a decrease in training accuracy. It can be inferred that the unit has caused over-fitting. Removing functional units and re-training is carried out iteratively until all the remaining functional units are necessary for the task. Table 6 shows the influences on perfor-

Table 5 The changes in test mIOU (%) when dropping functional units in layers conv6_1, conv6_2, conv6_3

	conv6_1		conv6_2		conv6_3	
	Train (%)	Test (%)	Train (%)	Test (%)	Train (%)	Test (%)
Complete	0.00	0.00	0.00	0.00	0.00	0.00
Unit 1	−0.33	−0.87	−0.14	−0.10	−0.14	−3.03
Unit 2	−0.02	−0.08	−0.09	−0.97	−0.09	−0.64
Unit 3	−0.05	−0.01	−1.04	−3.19	+0.00	+0.00
Unit 4	−0.58	−1.71	+0.00	−0.32	+0.00	+1.02
Unit 5	−1.11	−4.20	−0.01	+0.18	−0.29	−1.04
Unit 6	+0.00	+0.00	−0.36	−1.16	−0.03	−0.83
Unit 7	−0.01	−0.52	−0.01	−0.33	−0.44	−1.31
Unit 8	−0.14	+0.00	+0.00	+0.00	−0.41	−1.66

Table 6 The influences on capacity of models and performance brought by reducing the dependency and redundancy among convolutional kernels. (Performance is evaluated by mIOU (%)

Module/model	mIOU (%)	Number of parameters
Deeplab-V2 [50]	64.94	13.161e+7
Deep Lab-V2 [50] with capacity reduction	65.97	11.257e+7
The segmentation module [68]	65.07	6.559e+7
The simplified segmentation module	67.43	1.589e+7

mance brought by the methods of reducing the dependency and redundancy among convolutional kernels. The experiments are conducted on the segmentation module addressed in [68] as well as the Deeplab-V2 model [50]. The training data comes from the PASCAL VOC 2010 Person Part Dataset for body part segmentation [48, 78]. The dataset includes annotations on 3,533 images where 1,716 images are used for training. The ground truth labels are segmentation masks. The maximum number of iterations of training varies from 20,000 to 70,000. Training is terminated when the average training accuracy does not change in two consecutive epochs. Batch size is set to 6. As the number of feature maps reduces, regression needs more training iterations. The reduction in capacity is conducted in two steps: (1) Applying the above-mentioned methods to simplify the models/modules. The method for reducing redundancy is conducted firstly and the one for reducing dependency secondly. The former is suitable for simplifying a CNN which has trained on a small dataset and/or with great redundancy. The second method is suitable for simplifying a CNN which has no clear over-fitting problem or which has already been simplified by the first method. (2) Apply the teacher-student method addressed in Sect. 1 to train the small models on a much larger training data with the predictions of larger network as ground truth.

It can be inferred from Table 6 that the proposed methods for reducing the dependency and redundancy among kernels can be applied to improve performance and reduce the complexity of models. The removed functional units are only useful for performing segmentation on the images from the training data, but not useful for the segmentation on the test data. The test accuracy is measured after removing the redundant functional units and re-training.

The efficiency can be improved significantly by capacity optimization. Moreover, the reduction in time during inference over-weights the extra time spent on simplifying the model. This is due to the fact that training is conducted for once while inference is conducted for countless times.

4.2 Summary

In this section, three methods have been proposed to optimize the capacity of a CNN model for a human parsing task. The orthogonality among convolutional kernels, redundancy in convolutional layers and the dependency among kernels are studied to reduce the number of parameters in human parsing models. More importantly, the simplified models even slightly outperform original models. In other words, the capacity of a CNN model is optimized and a better convergence is achieved.

5 Conclusion and Future Work

In this chapter, we introduce the work on constructing a CNN with suitable capacity and architecture for a semantic segmentation task. We have discussed our three major contributions to architectural design and training of convolutional neural networks in the context of human parsing tasks. Firstly, we proposed novel training strategies and schemes for data augmentation. Secondly, we proposed the methods for evaluating the complexity of a segmentation task by analyzing different types of variances as well as the consistency in predictions from different models. Thirdly, we proposed three architectures of convolutional layers to reduce computational burdens while maintaining performance. This chapter echoes very well the theme of this volume "Deep Learning: Architectures, Algorithms, and Applications". Our future work will focus on the quantitative evaluation on the manifold of data which will help to guide the development of machine learning models.

References

1. Lee, J., Xiao, L., Schoenholz, S.S., Bahri, Y., Sohl-Dickstein, J., Pennington, J.: Wide neural networks of any depth evolve as linear models under gradient descent (2019). arXiv preprint arXiv:1902.06720
2. Rolnick, D., Tegmark, M.: The power of deeper networks for expressing natural functions. In: International Conference on Learning Representations (2018)
3. Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S.S., Pennington, J.: Dynamical isometry and a mean field theory of CNNs: how to train 10,000-layer vanilla convolutional neural networks (2018). arXiv preprint arXiv:1806.05393
4. Pierre, B., Roman, V.: Neuronal capacity. In: NIPS (2018)
5. Lei, N., Luo, Z., Yau, S.T., Gu, D.X.: Geometric understanding of deep learning (2018). arXiv preprint arXiv:1805.10451
6. Krizhevsky, A., Hinton, G.: Convolutional deep belief networks on cifar-10. In: Unpublished manuscript (2010)
7. Deng, J., Berg, A., Satheesh, S., Su, H., Khosla, A., Li, F.F.: Large scale visual recognition challenge 2012. Available: http://www.image-net.org/challenges/ILSVRC/2012/ (2012)
8. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
9. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition (2014). arXiv preprint arXiv:1409.1556
10. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going deeper with convolutions. In: IEEE Conference on Computer Vision and Pattern Recognition (2015)
11. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
12. Huang, G., Liu, Z., Weinberger, K.Q., van der Maaten, L.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, vol. 1. pp. 3 (2017)
13. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition, pp. 5987–5995 (2017)

14. Hu, J., Shen, L., Sun, G.: Squeeze-and-excitation networks (2017). arXiv preprint arXiv:1709.01507
15. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for scalable image recognition (2017). arXiv preprint arXiv:1707.07012
16. Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L.J., Li, F.F., Yuille, A., Huang, J., Murphy, K.: Progressive neural architecture search (2017). arXiv preprint arXiv:1712.00559
17. Real, E., Aggarwal, A., Huang, Y., Le, Q.V.: Regularized evolution for image classifier architecture search (2018). arXiv preprint arXiv:1802.01548
18. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning (2015)
19. He, K., Zhang, X., Sun, J.: Identity mappings in deep residual networks. In: European Conference on Computer Vision, pp. 630–645 (2016)
20. Simon, M., Rodner, E., Denzler, J.: ImageNet pre-trained models with batch normalization (2016). arXiv preprint arXiv:1612.01452, https://github.com/cvjena/cnn-models
21. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual networks (2015). Available: https://github.com/KaimingHe/deep-residual-networks
22. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual networks with 1 K layers (2016). Available: https://github.com/KaimingHe/resnet-1k-layers
23. He, K., Zhang, X., Ren, S., Sun, J.: Trained ResNet torch models (2016). Available: https://github.com/facebook/fb.resnet.torch/tree/master/pretrained
24. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT press (2016)
25. Kurakin, A., Goodfellow, I., Bengio, S., Dong, Y., Liao, F., Liang, M., Liang, J.: Adversarial attacks and defences competition (2018). arXiv preprint arXiv:1804.00097
26. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The PASCAL Visual object classes challenge 2012 (VOC2012) results. Available: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html (2012)
27. Luo, L., Xiong, Y., Liu, Y., Sun, X.: Adaptive gradient methods with dynamic bound of learning rate (2019). arXiv preprint arXiv:1902.09843
28. Anonymous: Shallow learning for deep networks. In: Under double-blind review (2018)
29. Zhou, Z.H.: A brief introduction to weakly supervised learning. Nat. Sci. Rev. **5**(1), 44–53 (2017)
30. Gong, K., Liang, X., Zhang, D., Shen, X., Lin, L.: Self-supervised structure-sensitive learning and a new benchmark for human parsing. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 932–940 (2017)
31. Gong, K., Liang, X., Li, Y., Chen, Y., Yang, M., Lin, L.: Instance-level human parsing via part grouping network. In: Proceedings of the European Conference on Computer Vision (2018)
32. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Zitnick, C.L.: Microsoft coco: common objects in context. In: Proceedings of ECCV (2014)
33. Neuhold, G., Ollmann, T., Rota Bulo, S., Kontschieder, P.: The mapillary vistas dataset for semantic understanding of street scenes. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 4990–4999 (2017)
34. Zhou, B., Zhao, H., Puig, X., Xiao, T., Fidler, S., Barriuso, A., Torralba, A.: Semantic understanding of scenes through the ADE20K dataset. Int. J. Comput. Vision 1–20 (2016)
35. Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Schiele, B.: The cityscapes dataset for semantic urban scene understanding. In: IEEE Conference on Computer Vision and Pattern Recognition (2016)
36. Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. In: Proceedings of IEEE CVPR (2014)
37. Uijlings, J., van de Sande, K., Gevers, T., Smeulders, A.: Selective search for object recognition. Int. J. Comput. Vision **104**(2), 154–171 (2013)
38. Li, J., Zhao, J., Wei, Y., Lang, C., Li, Y., Sim, T., Yan, S., Feng, J.: Multiple-human parsing in the wild (2017). arXiv:1705.07206
39. Long, J., Shelhamer, E., Darrell, T.: Fully convolutional networks for semantic segmentation. IEEE Trans. Pattern Anal. Mach. Intell. **39**(4), 640–651 (2017)

40. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: Available: http://www.pascal-network.org/challenges/VOC/voc2011/workshop/index.html (2011)
41. Li, X., Chen, H., Qi, X., Dou, Q., Fu, C.W., Heng, P.A.: H-DenseUNet: hybrid densely connected UNet for liver and liver tumor segmentation from CT volumes (2017). arXiv preprint arXiv:1709.07330
42. Kirillov, A., He, K., Girshick, R., Rother, C.: Panoptic segmentation (2018). arXiv preprint arXiv:1801.00868
43. de Geus, D., Meletis, P., Dubbelman, G.: Panoptic segmentation with a joint semantic and instance segmentation network (2018). arXiv preprint arXiv:1809.02110
44. Zheng, S.: Conditional random fields as recurrent neural networks. In: Proceedings of IEEE ICCV (2015)
45. Li, X., Zhao, L., Wei, L., Yang, M.H., Wu, F., Zhuang, Y., Ling, H., Wang, J.: DeepSaliency: multi-task deep neural network model for salient object detection. IEEE Trans. Image Process. **25**(8), 3919–3930 (2016)
46. Xia, F., Wang, P., Chen, X., Yuille, A.L.: Joint multi-person pose estimation and semantic part segmentation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2017)
47. Jiang, Y., Chi, Z.: A fully-convolutional framework for semantic segmentation. In: Proceedings of IEEE DICTA (2017)
48. Chen, X., Mottaghi, R., Liu, X., Fidler, S., Urtasun, R., Yuille, A.L.: Detect what you can: detecting and representing objects using holistic models and body parts. In: Proceedings of IEEE CVPR (2014)
49. Chen, L.C., Barron, J.T., Papandreou, G., Murphy, K., Yuille, A.L.: Semantic image segmentation with task-specific edge detection using cnns and a discriminatively trained domain transform. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2016)
50. Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Alan, L.: Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 834–848 (2018)
51. Chen, L.C., Papandreou, G., Schroff, F., Adam, H.: Rethinking atrous convolution for semantic image segmentation (2017). arXiv preprint arXiv:1706.05587
52. Li, Q., Arnab, A., Torr, P.H.: Weakly-and semi-supervised panoptic segmentation. In: Proceedings of the European Conference on Computer Vision, pp. 102–118 (2018)
53. Dai, J., He, K., Sun, J.: Boxsup: exploiting bounding boxes to supervise convolutional networks for semantic segmentation. In: Proceedings of IEEE ICCV (2015)
54. Zhang, L., Yang, Y., Gao, Y., Yu, Y., Wang, C., Li, X.: A probabilistic associative model for segmenting weakly supervised images. IEEE Trans. Image Process. **23**(9), 4150–4159 (2014)
55. Zamir, A.R., Sax, A., Shen, W., Guibas, L.J., Malik, J., Savarese, S.: Taskonomy: disentangling task transfer learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (2018)
56. Donahue, J., Krahenbuhl, P., Darrell, T.: Adversarial feature learning (2016). arXiv preprint arXiv:1605.09782
57. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Massachusetts, Cambridge (2016)
58. Wang, Y.X., Ramanan, D., Hebert, M.: Growing a brain: fine-tuning by increasing model capacity. In: Proceedings of IEEE CVPR (2017)
59. Wang, Y., Xie, L., Liu, C., Qiao, S., Zhang, Y., Zhang, W., Yuille, A.L.: Sort: second-order response transform for visual recognition. In: Proceedings of IEEE ICCV (2017)
60. Sigaud, O., Droniou, A.: Towards deep developmental learning. IEEE Trans. Cognit. Dev. Syst. **8**(2), 99–114 (2016)
61. Tishby, N., Zaslavsky, N.: Deep learning and the information bottleneck principle. In: Information Theory Workshop (ITW) (2015)
62. Gabrié, M., Manoel, A., Luneau, C., Barbier, J., Macris, N., Krzakala, F., Zdeborová, L.: Entropy and mutual information in models of deep neural networks (2018). arXiv preprint arXiv:1805.09785

63. Guss, W.H., Salakhutdinov, R.: On characterizing the capacity of neural networks using algebraic topology (2018). arXiv preprint arXiv:1802.04443
64. Zagoruyko, S., Komodakis, N.: Wide residual networks (2016). arXiv preprint arXiv:1605.07146
65. Krizhevsky, A., Hinton, G.E.: Learning Multiple Layers of Features from Tiny Images. Toronto (2009)
66. Gong, K., Liang, X., Zhang, D., Shen, X., Lin, L.: Look into person: self-supervised structure-sensitive learning and a new benchmark for human parsing. In: IEEE Conference on Computer Vision and Pattern Recognition (2017)
67. Chen, L.C., Papandreou, G., Kokkinos, I., Murphy, K., Yuille, A.L.: Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE Trans. Pattern Anal. Mach. Intell. **40**(4), 834–848 (2018)
68. Jiang, Y., Chi, Z.: A CNN model for semantic person part segmentation with capacity optimization. IEEE Trans. Image Process. (2018)
69. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network (2015). arXiv preprint arXiv:1503.02531
70. Cao, Z., Hidalgo, G., Simon, T., Wei, S., Sheikh, Y.: Openpose: realtime multi-person 2D pose estimation using part affinity fields (2018). arXiv preprint arXiv:1812.08008
71. Zhang, R., Phillip, I., Alexei, A.E.: Colorful image colorization. In: European Conference on Computer Vision (2016)
72. Redmon, J., Farhadi, A.: YOLO9000: better, faster, stronger (2017). arXiv preprint arXiv:1612.08242
73. Zhao, H., Shi, J., Qi, X., Wang, X., Jia, J.: Pyramid scene parsing network. In: IEEE Conference on Computer Vision and Pattern Recognition (2017)
74. Andriluka, M., Pishchulin, L., Gehler, P., Schiele, B.: 2d human pose estimation: new benchmark and state of the art analysis. In: Proceedings of IEEE CVPR (2014)
75. Huang, L., Liu, X., Lang, B., Yu, A.W., Wang, Y., Li, B.: Orthogonal weight normalization: solution to optimization over multiple dependent stiefel manifolds in deep neural networks. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
76. Desjardins, G., Simonyan, K., Pascanu, R., Kavukcuoglu, K.: Natural neural networks. In: Neural Information Processing Systems (2015)
77. Abdi, H., Williams, L.J.: Principal component analysis. Wiley Interdiscip. Rev. Comput. Stat. **2**(4), 433–459 (2010)
78. Everingham, M., Eslami, S.A., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The pascal visual object classes challenge: a retrospective. Int. J. Comput. Vision **111**(1), 98–136 (2015)
79. Deng, J., Berg, A., Satheesh, S., Su, H., Khosla, A., Li, F.F.: Available: http://www.image-net.org/challenges/LSVRC/2012/ (2012)
80. Nowlan, S.J., Hinton, G.E.: Simplifying neural networks by soft weight-sharing. Neural Comput. **4**(4), 473–493 (1992)
81. Moon, T.K.: The expectation-maximization algorithm. IEEE Signal Process. Mag. **13**(6), 47–60 (1996)

Using Convolutional Neural Networks to Forecast Sporting Event Results

Mu-Yen Chen, Ting-Hsuan Chen and Shu-Hong Lin

Abstract Sporting events like the FIFA World Cup and the World Baseball Classic have increased in popularity, and the enthusiasm with which these competitions are reported and commented on is evidence of their wide-reaching influence. These games are popular discussion topics. Many who follow sports are only casual fans, but even these "temporary" fans have the same expectations that all fans do: the teams they support should be able to win. This study selects the National Basketball Association (NBA) to represent competitive ball sports. Predictions herein are based on the examination and statistical analysis of previous records of NBA games. This research integrates the field of sports outcome predictions with the methodology of deep learning via convolutional neural networks. Training and predictions are modelled on statistics gleaned from a total of 4,235 games over the past three years. We analyze the training results of a variety of model structures. While previous studies have applied convolutional neural networks to image or object recognition, our study proposes a specific encoding method that is integrated with deep learning in order to predict the results of future games. The prediction accuracy rate of the model herein is approximately 91%, while the deviation is approximately 0.2. These strong results confirm the validity of our designated encoding method.

Keywords Sports prediction · Deep learning · Convolutional neural networks · National basketball association (NBA)

M.-Y. Chen (✉) · S.-H. Lin
Faculty of Information and Management, National Taichung University of Science and Technology, Taichung, Taiwan
e-mail: mychen.academy@gmail.com

S.-H. Lin
e-mail: abc9652000@gmail.com

T.-H. Chen
Faculty of Finance, National Taichung University of Science and Technology, Taichung, Taiwan
e-mail: thchen@nutc.edu.tw

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_9

1 Introduction

Whenever the FIFA World Cup or the World Baseball Classic tournaments begin, the number of devoted or casual fans around the globe rises significantly throughout the season. Online search trends in connection with the games also surge sharply. Based on the hot search terms shown in Google's Trends service, it is obvious that competitive sports are of great interest to most people, especially during international tournament seasons. Figure 1 illustrates a Google hot trend for the search term "soccer" in July 2018 [1], showing how the search trend soared upward in the middle of June 2018 in connection with the FIFA World Cup.

Business Weekly [2] reported that the Hokkaido Nippon-Ham Fighters, a professional baseball team in Japan, scouted and identified the most suitable players for each position by using the Baseball Operation System (BOS) to analyze its player's capabilities. Sports vary in terms of types, rules and skills, and are also subject to various factors that are difficult to quantify (e.g., weather conditions, morale, etc.). In order to exclude a variety of unstable factors, this study selected the games of the National Basketball Association (NBA) as the subject of analysis, because they are less affected by factors that cannot be controlled, and the data are publicly available. Fong [3] predicted the outcomes and scores of Major League Baseball (MLB) games. However, since too many variables were taken into consideration, the model results tended to fail to converge, reducing the accuracy of the predictions. Therefore, we chose NBA games, which are subject to fewer determinants. The NBA is also the most prestigious of the professional basketball associations.

The concept of deep learning has gained ground rapidly in recent years. After the computer Alpha Go defeated South Korean Go grandmaster Lee Sedol, deep learning became even more well known around the world. However, Alpha Go was soon superseded by AlphaGo Zero, which defeated Alpha Go when the two programs played against each other. This shows how rapidly deep learning has been developed and optimized.

While deep learning has been applied in studies of image/object recognition (e.g., board game notations, paintings, people/things/objects in pictures, etc.), deep learning has also been commonly applied to feature extraction. Rarely have previous researchers used the deep learning method to predict the outcomes of competitive sports. Such outcomes are more commonly predicted via the algorithms of statistical regression analysis, artificial neural networks, or support vector machines. Thus, this

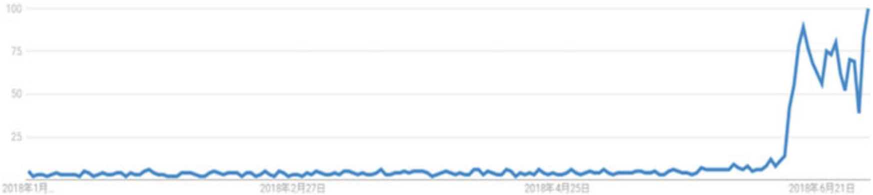


Fig. 1 Google hot trend (Keyword: *Soccer* in Chinese)

study has adopted the deep learning method to predict the outcomes of competitive sports.

We use the records of NBA games from the past three years as the subject of analysis and examination, and we model them in accordance with the deep learning method. The objectives of this research are as follows:

- To reduce the amount of data required for predicting the outcomes.
- To convert the format of the original data to make it a proper fit for deep learning.
- To compare the accuracy of the prediction results from models with different structures.

2 Literature Review

2.1 Convolutional Neural Network Architecture

A convolutional neural network is a class of feed-forward artificial neural networks. Its neurons can respond to peripheral units within a specific area of coverage. Its application to visual imagery has yielded excellent results. The structure of the convolutional neural network mimics the neurons in the biological brain. Hubel and Wiesel [4] first noticed that when a cat is visually observing different things, the neurons in the brain react differently. Fukushima [5] proposed an unsupervised conceptual network, which was able to learn geometrical variations in graphics on its own.

Indolia et al. [6] presented the architecture of the convolutional neural network in detail. The convolutional neural network is mainly composed of several structures: the input layer, convolutional layers, pooling layers, activation function, fully connected layers and loss function, and the output layer. This architecture is shown in Fig. 2.

The main structure in a convolutional neural network consists of one or more convolutional layers with a fully connected layer at the end. It also includes the associated weights and the results derived from pooling layers. This structure enables

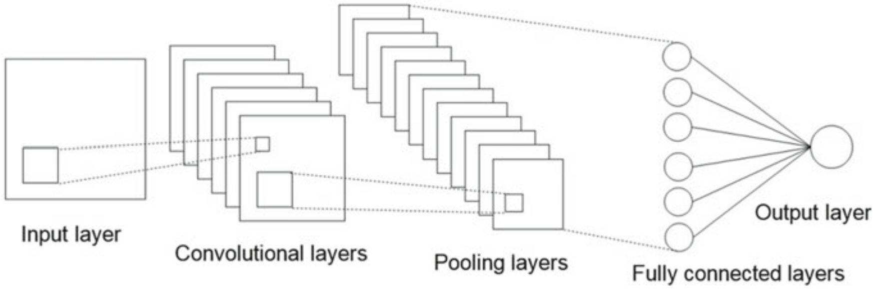


Fig. 2 Architecture of convolutional neural network

the convolutional neural network to make use of the two-dimensional structural characteristics of the data. The convolutional neural network can be trained via back propagation, and thus needs fewer parameters than other deep networks.

2.2 Related Research Regarding Sports Predictions

Research regarding prediction requires access to historical data. Feature extraction from existing data allows a future target to be predicted. Sports predictions are no exception.

Craig et al. [7] proposed the relationship between a team's offense and the ambient temperature in American football games, and analyzed it using multi-level regression analysis. The adopted dataset was provided by the National Football League (NFL). The results showed that the level of aggression of the football team's offense has a positive linear relationship with the ambient temperature. The higher the ambient temperature, the greater the intensity with which the offense plays. However, this applied to home teams only.

Maszczyk et al. [8] used four factors to predict javelin throw distances: the first-step distance in throwing the javelin, the strength of the body and arms, the strength of the abdominal muscles, and the grip force of the hand. A regression model and an artificial neural network model were established. Eventually, the differences between their results were compared. The deviation of the artificial neural network was 16.77 m on average, and that of the regression model was 29.45 m on average, indicating that the artificial neural network had much better predictive capability. This also proved that the prediction results of the artificial neural network were better than those of the widely used regression model.

Bačić [9] used embedded sensors to test a golf swing application. The golf player's current physiological state and the flight path of the golf ball were recorded for each swing. These data were quantified via HMMA (Human Motion Modeling and Analysis). The golf ball paths were predicted using the swing lengths and the attack angles.

Bunker and Thabtah [10] noted that machine learning was a very promising method in the domains of classification and prediction. The large amounts of money involved in wagers on sporting events demand highly precise predictions. The researchers rigorously reviewed the literature on machine learning. The artificial neural network was then adopted as a novel framework by which machine learning can be used as a learning strategy.

Kipp et al. [11] established an artificial neural network model to predict the net joint moments (NJM) of the hip, knees and ankles when a weightlifter is lifting weights. Data from seven collegiate weightlifters were applied to training and testing. Although the prediction performance levels were lower than those in previous literature, the barbell motion and the time series data of NJM were similar to those of past high-level athletes. This also proved that the prediction results were quite stable.

The extant literature has shown that research into any sort of prediction is heavily dependent on historical data. Such data may also have a large number of columns. To predict well, important features must be extracted. This study uses NBA data. After selecting more important columns from the data, a convolutional neural network is used to conduct feature extraction and model training.

3 Research Methods

3.1 Development Environment

The hardware for this research was a PC equipped with a 64-bit Win7 operating system, with a 4-core CPU and 8 GB RAM. We used the Python programming language to collect and preprocess game data. We also used an Oracle VM VirtualBox virtual machine for deep learning development and model establishment. The operating system was Linux and the programming language was also Python, with a Pytorch framework.

3.2 Research Process

This study's research process can be divided into five major steps: data collection, data preprocessing, data encoding, data modeling, and performance evaluation. The research model and process is illustrated in Fig. 3.

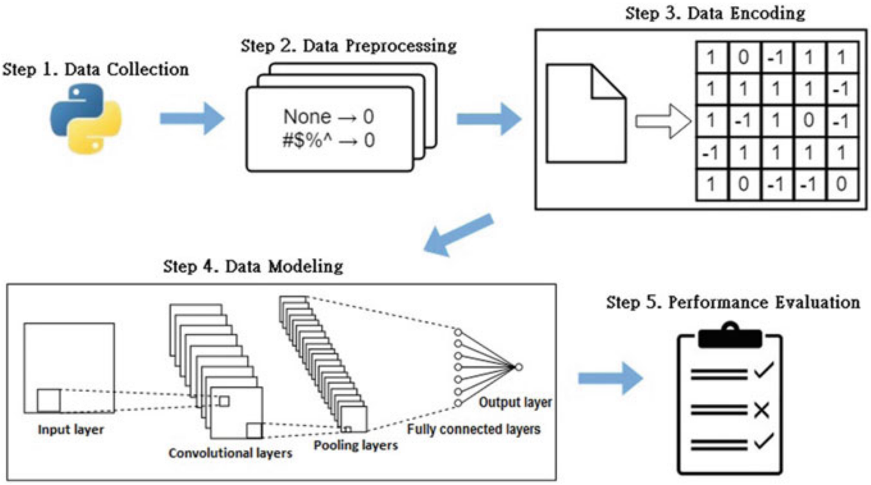
- Step 1: Data Collection

The NBA game data was downloaded for analysis from the NBA Taiwan official website. Figure 4 shows a sample of the original data [12]. Python was used as a data crawler, collecting game records from the October 2014 to April 2017 time period, which comprised the past three NBA seasons for a total of 4,235 games. Home teams had 2,473 wins and 1,762 losses during this period.

- Step 2: Data Preprocessing

Once the Python data crawler had acquired the data, we found that differences in the numbers of players in many games, as well as non-starters, resulted in missing values and different numbers of columns. In the data processing stage, the null values in data fields were filled with a value of 0 to complete the game data for experimental purposes. We gathered a total of 20 data items for each player, as shown in Table 1.

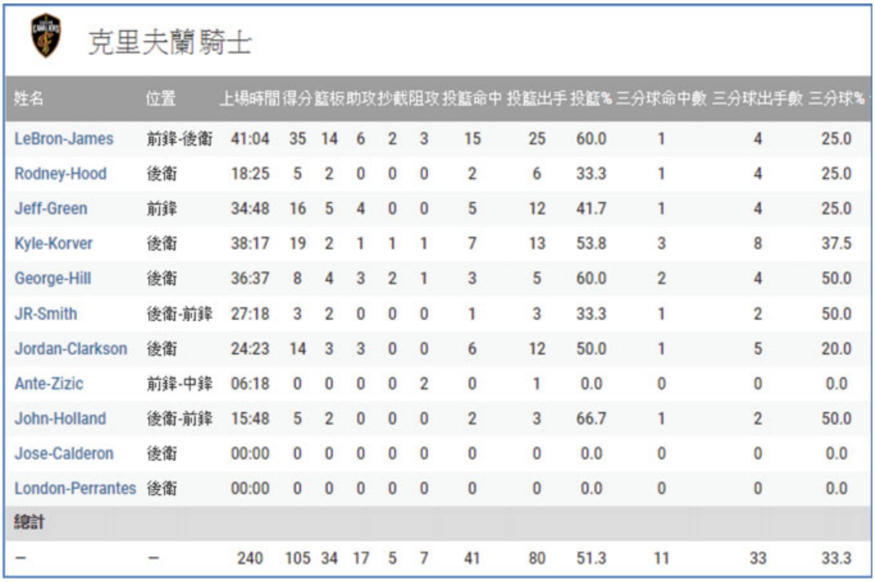
In 1989, Manley [13] proposed the term EFF (efficiency) to calculate NBA player performance. EFF was calculated based on five additive data items (points, rebounds, assists, steals and blocks) and three subtractive items (missed field goals, missed free throws, and turnovers). We chose to include only the items that had more influence on



1	0	-1	1	1
1	1	1	1	-1
1	-1	1	0	-1
-1	1	1	1	1
1	0	-1	-1	0

Fig. 3 Research model and process

the overall game; thus, turnovers were not factored into the calculation. Regarding the more influential items in terms of final points, the point calculation for this study included free throws and three point field goals, which were calculated as percentages. The percentages of field goals, three point field goals, and free throws were each calculated by dividing the number of shots made by the number of shots



克里夫蘭騎士

姓名	位置	上場時間	得分	籃板	助攻	抄截	阻攻	投籃命中	投籃出手	投籃%	三分球命中數	三分球出手數	三分球%
LeBron-James	前鋒-後衛	41:04	35	14	6	2	3	15	25	60.0	1	4	25.0
Rodney-Hood	後衛	18:25	5	2	0	0	0	2	6	33.3	1	4	25.0
Jeff-Green	前鋒	34:48	16	5	4	0	0	5	12	41.7	1	4	25.0
Kyle-Korver	後衛	38:17	19	2	1	1	1	7	13	53.8	3	8	37.5
George-Hill	後衛	36:37	8	4	3	2	1	3	5	60.0	2	4	50.0
JR-Smith	後衛-前鋒	27:18	3	2	0	0	0	1	3	33.3	1	2	50.0
Jordan-Clarkson	後衛	24:23	14	3	3	0	0	6	12	50.0	1	5	20.0
Ante-Zizic	前鋒-中鋒	06:18	0	0	0	0	2	0	1	0.0	0	0	0.0
John-Holland	後衛-前鋒	15:48	5	2	0	0	0	2	3	66.7	1	2	50.0
Jose-Calderon	後衛	00:00	0	0	0	0	0	0	0	0.0	0	0	0.0
London-Perrantes	後衛	00:00	0	0	0	0	0	0	0	0.0	0	0	0.0
總計													
–	–	240	105	34	17	5	7	41	80	51.3	11	33	33.3

Fig. 4 The NBA data record of the home team in a single game

Table 1 NBA game data columns

Title of column	Title of column	Title of column	Title of column
Minutes played	Blocks	3 point field goals attempted	Offensive rebounds
Points	Field goals made	3 point field goals percentage	Defensive rebounds
Rebounds	Field goals attempted	Free throws made	Turnovers
Assists	Field goal percentage	Free throws attempted	Fouls
Steals	3 point field goals made	Free throw percentage	Positive/Negative (±)

attempted. Since the numbers of shots made and attempted were already included in the percentages, these two items were excluded from the calculation. On the other hand, since the number of rebounds included offensive and defensive rebounds, these two items were also excluded from the calculation. Since the efficiency value did not include the number of fouls, this study also excluded that data.

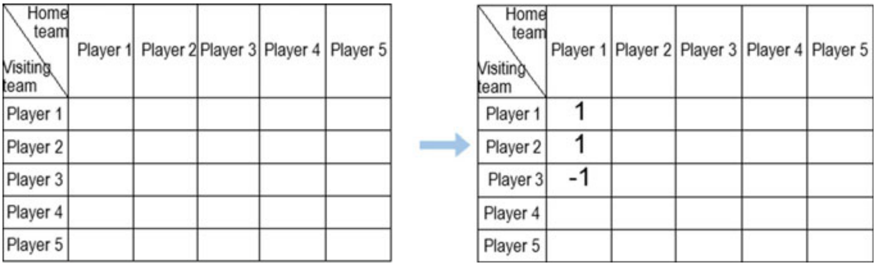
The positive and negative values (±) were the player's impact on the score when on the court. When Player A was on the court, and he or a teammate scored 2 points, the value was a +2. Lost points were subtracted directly. The positive and negative values were attached to Player A until he was substituted or the game ended. The positive and negative values already accounted for the influence of time, so the number of minutes player was on the court was not included in the calculation. Table 2 shows the data columns and their calculation methods as used in this research.

Since there was no measuring standard for the final score and statistics in each match, and overtimes would result in different lengths of time in a typical match, the maximum values of the statistics might affect the calculated results. Thus, data needed to be normalized afterwards. Normalization was conducted on a game-by-game basis by calculating the level of influence of each statistic of each player in a single game. The minimum value was 0 and the maximum value was 1. The normalization method is shown in Eq. (1).

$$y = \frac{\mathrm{x} - \min(\mathrm{x})}{\max(\mathrm{x}) - \min(x)} \tag{1}$$

Table 2 Game data columns used in this research

Title of columns	Calculation method	Title of columns	Calculation method
Points	Additive item	Field goal percentage	Additive item
Rebounds	Additive item	3 point field goals percentage	Additive item
Assists	Additive item	Free throw percentage	Additive item
Steals	Additive item	Turnovers	Subtractive item
Blocks	Additive item	Positive and negative values	Additive item



Home team / Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1					
Player 2					
Player 3					
Player 4					
Player 5					

Home team / Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1	1				
Player 2	1				
Player 3	-1				
Player 4					
Player 5					

Fig. 5 Schematic diagram of the matrix encoding process

- Step 3: Data Encoding

Since the convolutional neural network has often been used in studies on image recognition, the corresponding data types did not match the data type of the present research. At this stage, the player's comprehensive skill performance levels calculated after normalization in Step 2 were encoded to conform to the data type of the convolutional neural network for the benefit of model construction.

The extent of each player's influence with respect to each column in a single game was already calculated in Step 2. Subsequently, those normalized columns were calculated. The calculated result could be regarded as the comprehensive performance evaluation of each player in a single game. With that, the home and visiting team players were compared one by one, and the comparison results were encoded. The comprehensive performance value of each player was compared separately from the perspective of the home team. If the influence of the home team player was greater than that of the visiting team player, the encoded value was 1; otherwise it was -1. If the extent of player influence was equivalent, it was encoded as 0. The encoding process is shown in Fig. 5. The NBA allows a maximum of 16 players to be registered in a given match; hence, the encoded matrix size was 16×16.

- Step 4: Data Modeling

The input data for modeling was the matrix data encoded in Step 3. It was a two-dimensional 16×16 matrix of the final target field, i.e., win and loss. Win or loss was determined from the perspective of the home team. The result of the game was encoded as a 1 for a win, and a 0 for a loss.

We then established a convolutional neural network model with different structures and parameters for the experiments. We expected to construct a better model by adjusting parameters. Four models of different architectures were established in this research. To prevent model overfitting, a dropout layer (setup value 10%) was set after the pooling layer. The detailed architecture settings of the models are shown in Table 3. These four models conducted 200 training iterations of 10 cross-validations. Other detailed parameter settings of the models are shown in Table 4.

Table 3 CNN architectures under different structures

CNN model architecture	Model 1	Model 2	Model 3	Model 4
Convolutional layer 1	2 × 2	2 × 2	2 × 2	5 × 5
Pooling layer 2	2 × 2	5 × 5	2 × 2	2 × 2
Dropout setting	10%	10%	10%	10%
Convolutional layer 2	2 × 2	2 × 2	2 × 2	2 × 2
Pooling layer 2	2 × 2	2 × 2	2 × 2	2 × 2
Dropout setting	10%	10%	10%	10%
Convolutional layer 3	–	–	2 × 2	–
Pooling layer 3	–	–	2 × 2	–
Dropout setting	–	–	10%	–
Fully connected layer	1	1	1	1

Table 4 Parameter settings of the CNN model

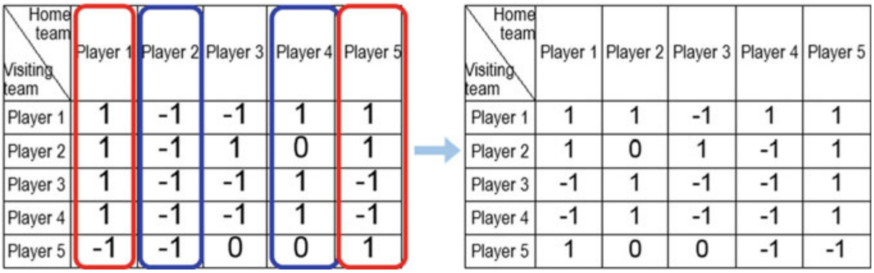
	Parameter values
Training iterations	200
Batch size	32
Learning rate	0.0015
Activation function	ReLU
Loss value calculation	Cross entropy
Optimizer	Adam

- Step 5: Performance Evaluation

The results of the convolutional neural network were evaluated based on different architectures, parameters and datasets. The results produced by using different datasets for experiments were also reviewed and analyzed. We compared the various results to the artificial neural network to analyze the differences.

3.3 Experiment Design

In addition to the architectural differences, other parameters were also adjusted to optimize the results of the convolutional neural network. We compared the sizes between the convolutional layer and the convolution filter in the pooling layer, and we also adjusted the batch size from 32 to 512. The learning rate was set to 0.0015. The loss value was calculated via cross entropy. We used the Adam optimizer, which is more commonly used than the stochastic gradient descent method and can also update the weights in a more stable manner.



Home team / Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1	1	-1	-1	1	1
Player 2	1	-1	1	0	1
Player 3	1	-1	-1	1	-1
Player 4	1	-1	-1	1	-1
Player 5	-1	-1	0	0	1

Home team / Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1	1	1	-1	1	1
Player 2	1	0	1	-1	1
Player 3	-1	1	-1	-1	1
Player 4	-1	1	-1	-1	1
Player 5	1	0	0	-1	-1

Fig. 6 Horizontally flipped matrix

- Experiment 1

The data content used in Experiment 1 was a two-dimensional matrix encoded by the coding method of this research. The experiments were performed using the aforementioned convolutional neural network model with different architectures. The data content contained 4,235 records.

- Experiment 2

The data content used in Experiment 2 was a two-dimensional matrix created by horizontally flipping Dataset 1. The horizontal flipping concept is depicted in Fig. 6. The left side of Fig. 6 is the already encoded two-dimensional matrix, while the right side of Fig. 6 is the matrix result after reversing horizontally.

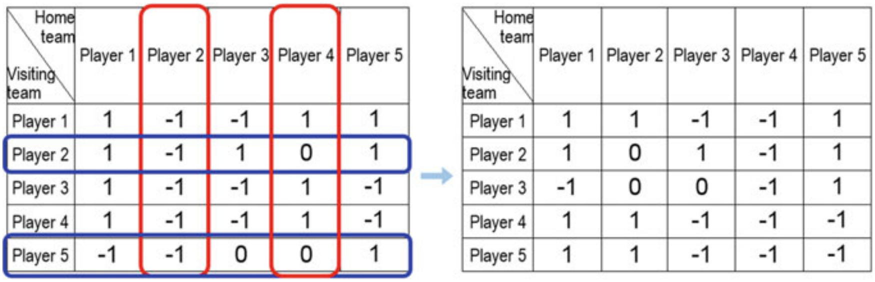
- Experiment 3

The data content used in Experiment 3 was composed of Datasets 1 and 2 so that a large amount of data could be used to train the convolutional neural network. We expected to be able to improve the performance of the model by increasing the data amount. Dataset 3 had a total of 8,470 entries, the largest data volume of the three sets.

- Experiment 4

The data used in Experiment 4 was processed by randomly swapping the encoded two-dimensional matrix by rows and columns. The concept of random swapping is illustrated in Fig. 7. The left side of Fig. 7 shows the encoded two-dimensional matrix, in which the red boxes are the columns to be swapped, and the blue boxes are the rows to be swapped. The right side of Fig. 7 shows the result after random swapping. Dataset 4 had 4,235 entries.

After experiments were conducted with convolutional neural network models of different architectures and parameters, in addition to the comparisons between each model, the results were compared with those of the back propagation neural network. Weka ver3.8 was used to establish the back propagation neural network environment.



Home team Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1	1	-1	-1	1	1
Player 2	1	-1	1	0	1
Player 3	1	-1	-1	1	-1
Player 4	1	-1	-1	1	-1
Player 5	-1	-1	0	0	1

Home team Visiting team	Player 1	Player 2	Player 3	Player 4	Player 5
Player 1	1	1	-1	-1	1
Player 2	1	0	1	-1	1
Player 3	-1	0	0	-1	1
Player 4	1	1	-1	-1	-1
Player 5	1	1	-1	-1	-1

Fig. 7 Randomly sequenced matrix

Table 5 Confusion matrix

Prediction reality	Positive	Negative
Positive	TP (True Positive)	FN (False Negative)
Negative	FP (False Positive)	TN (True Negative)

3.4 *Performance Evaluation*

This research used the accuracy and loss of the final experimental prediction as the evaluation criteria. A confusion matrix was used to calculate accuracy. As shown in Table 5, Eq. (2) was used to compute accuracy. Loss was calculated via cross entropy [14]. This computation is shown in Eq. (3). When the loss approached 0, the result of the model prediction was closer to the actual result. Through the loss values, the model could also perform back propagation to modify the weights and parameters inside the model. The loss value did not fully represent the quality of the model. It could only determine whether the prediction result of the model was close to the actual values.

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{2}$$

$$H(p, q) = \mathrm{I} \sum_{x} p_{\mathrm{x}} \log q_x \tag{3}$$

4 Experiment Results

4.1 *Dataset Description*

This research used four datasets with different contents to conduct the experiments. The use of four datasets was intended to confirm the feasibility of our proposed encoding methods. The contents of these four datasets are described respectively, below. Table 6 shows the combination of the content and the purpose of each dataset.

4.2 Results of Experiments 1 and 2

This section shows the experimental results of the four models with different architectures after using Datasets 1 and 2 under the conditions of different batch sizes. The data contents of Datasets 1 and 2 were the already encoded two-dimensional matrix data. The experimental results are shown in Tables 7 and 8; the contents are the average of the results after conducting 10-fold cross-validation experiments.

The accuracy of the results of Experiments 1 and 2 were very close. Their accuracy rates were approximately 90%, but Model 2 was slightly lower at about 89%. The difference between Models 1 and 2 was the size of the pooling layer. In the pooling layer of Model 2, a 5 × 5 convolution filter was used for sampling. The reason for the poor result might be that the two-dimensional matrix used in this research was 16 × 16. Compared with the pixel matrix of a photo, the matrix of this research is indeed too small. Direct oversized sampling could have destroyed the structure of the small matrix, resulting in feature losses that eventually affected the accuracy. Although

Table 6 Description of each dataset

	Dataset 1	Dataset 2	Dataset 3	Dataset 4
Purpose of experiment	Confirm the feasibility of the encoding method proposed in this research	Expect to retain its feature values after the matrix is flipped	Expect to enhance model performance via increasing data amount	Expect the encoding method of this research will not be affected by the arrangement order
Data composition	The matrix data of the encoding method proposed in this research	The result created by horizontally flipping Dataset 1	The summation of Dataset 1 and Dataset 2	The matrix created by randomly swapping the sequence
Data amount	4,235	4,235	8,470	4,235

Table 7 Experimental results with 10-fold cross-validation in Experiment 1

Batch size	Experimental result	Model 1	Model 2	Model 3	Model 4
32	Accuracy	90.24%	89.35%	89.80%	89.80%
	Loss	0.206	0.213	0.192	0.224
64	Accuracy	90.62%	**89.49%**	90.02%	90.01%
	Loss	0.206	0.215	0.212	0.216
128	Accuracy	90.90%	89.17%	90.36%	90.34%
	Loss	0.209	0.225	0.198	0.222
256	Accuracy	**91.10%**	89.41%	**90.51%**	90.57%
	Loss	0.201	0.226	0.202	0.194
512	Accuracy	90.98%	89.17%	90.37%	**90.65%**
	Loss	0.198	0.237	0.196	0.186

Table 8 Experimental results with 10-fold cross-validation in Experiment 2

Batch size	Experimental result	Model 1	Model 2	Model 3	Model 4
32	Accuracy	90.51%	89.21%	90.08%	89.92%
	Loss	0.225	0.227	0.233	0.228
64	Accuracy	90.72%	89.38%	89.94%	89.96%
	Loss	0.239	0.214	0.217	0.232
128	Accuracy	90.78%	89.26%	90.20%	90.20%
	Loss	0.213	0.246	0.229	0.228
256	Accuracy	91.02%	**89.51%**	90.40%	90.52%
	Loss	0.208	0.227	0.232	0.208
512	Accuracy	**91.22%**	89.31%	**90.52%**	**90.95%**
	Loss	0.205	0.233	0.229	0.214

the size of the convolution filter in Model 4 was the same as in Model 2, the sampling method and the sequence were different in the convolutional layer; hence the matrix structure was not likely to be destroyed, ensuring a more accurate prediction. Model 3 was the deepest model. Normally, performance is better when there are more convolutional and pooling layers. There are two reasons for the setbacks. The first reason is insufficient data volume. Deep learning requires a large amount of data in order to raise the performance level of the model. The approximately 4,000 entries in our dataset were insufficient in terms of data volume. The second reason is insufficient matrix size. The convolutional neural network model was originally applied to image sampling, and digital images can often measure more than 16 pixels on a side. Taking the commonly used handwriting recognition MNIST dataset as an example, the image size is 28 × 28 pixels, not to mention the much larger image sizes used in other studies or in videos.

Increasing the batch size is likely to reduce the time required to train the model. The loss value and accuracy would also be improved because the deviations could be more easily identified in a larger batch size, and the loss value could hence be lowered. As shown in the results, the batch size must be adjusted according to the structure and data characteristics of the model.

4.3 Results of Experiment 3

This section shows the experimental results of the four models after using Dataset 3 under the conditions of different batch sizes. Dataset 3 was the summation of Datasets 1 and 2. The training data volume is also extremely important to deep learning. Hence the datasets of Experiments 1 and 2 were summed up and trained in Experiment 3. The experimental results are depicted in Table 9.

Table 9 Experimental results with 10-fold cross-validation in Experiment 3

Batch size	Experimental result	Model 1	Model 2	Model 3	Model 4
32	Accuracy	72.80%	**71.43%**	**72.32%**	71.84%
	Loss	0.451	0.461	0.477	0.478
64	Accuracy	72.97%	71.37%	72.25%	**71.97%**
	Loss	0.464	0.468	0.456	0.473
128	Accuracy	73.11%	71.18%	72.13%	71.81%
	Loss	0.459	0.464	0.471	0.471
256	Accuracy	73.03%	71.29%	72.24%	71.57%
	Loss	0.457	0.463	0.459	0.485
512	Accuracy	**73.19%**	71.07%	72.01%	71.53%
	Loss	0.455	0.475	0.472	0.479

The accuracy of Experiment 3 was about 70%; the loss value rose to 0.4 or even 0.5. Compared with Experiments 1 and 2, the accuracy and loss value were rather different. Since Experiment 3 used twice the data volume for training, the results were expected to be much better, and it took much more time to train.

After comparing the results of Experiment 3 with those of Experiments 1 and 2, it was determined that the integrated dataset had failed to accurately predict the results because of the different features of the datasets. In other words, when Dataset 1 was flipped as Dataset 2, the results was two datasets that were completely different from each other. As shown in the result of Experiment 3, the flipped matrix could not retain the characteristics of the original images. In object recognition, the target for identification is usually symmetrical in terms of left and right; i.e., after flipping, the relative positional relationships of the features can still be found. However, in Experiment 3, such a relationship could not be found. This made accurate predictions difficult in Experiment 3.

4.4 Results of Experiment 4

This section shows the experimental results of the four models after using Dataset 4 under the conditions of different batch sizes. The purpose of Experiment 4 was to determine whether after randomly swapping the matrix with a precondition, the dataset could retain certain characteristics in order for the models to extract the feature values. The results of Experiment 4 are shown in Table 10.

The results of Experiment 4 were similar to those of Experiments 1 and 2. Although the results were not as good as Experiments 1 and 2, the accuracy of each model was about 86%, and the loss value was about 0.25. The results of Experiment 4 were very close. The differences between models were nearly identical to those of Experiments 1 and 2. The reasons were also the same.

With respect to the significance of randomly swapping the two-dimensional matrix in Experiment 4, it should be noted that no matter how the matrix contents were randomly swapped, only the rows and columns of the two-dimensional matrix were used as units to perform random swapping, rather than completely randomizing the entire two-dimensional matrix. In other words, regardless of the arrangement sequence of each player in the home and visiting teams, the players were compared one by one eventually. Though not as good as in Experiment 1, the prediction results in Experiment 4 were pretty good, and the prediction capability was rather stable.

4.5 Discussion

A total of four different datasets, four different convolutional neural network architectures, and five different batch sizes were used to verify the feasibility of the encoding methods proposed in this research. Only Experiment 3 failed to achieve better performance. Experiments 1, 2, and 4 achieved a good level of performance. Among the four experiments, Model 1 stood out most prominently. In Table 11 shows the best results of Model 1 in Experiments 1 to 4.

We compared the foregoing with the artificial neural network, which was established using Weka version 3.8. The setup parameters are shown in Table 12. The artificial neural network used Dataset 1, for which the accuracy rate was about 91.64%. On the other hand, the best accuracy in this research was approximately 91.78%.

The encoding method proposed in this research for the convolutional neural network worked better than the artificial neural network. We found that, after encoding the data, the regional feature extraction method of the convolutional neural network could effectively extract features. Through the coding method proposed in this research, the calculated values were used to compare and analyze the relative capabilities of the home and visiting team players. These values were then encoded

Table 10 Experimental results with 10-fold cross-validation in Experiment 4

Batch size	Experimental result	Model 1	Model 2	Model 3	Model 4
32	Accuracy	86.07%	85.59%	85.65%	85.09%
	Loss	0.235	0.280	0.256	0.259
64	Accuracy	86.43%	85.72%	85.93%	85.63%
	Loss	0.239	0.285	0.261	0.249
128	Accuracy	86.61%	85.74%	85.77%	85.71%
	Loss	0.247	0.284	0.260	0.244
256	Accuracy	86.81%	85.69%	85.84%	85.89%
	Loss	0.240	0.282	0.266	0.259
512	Accuracy	**87.12%**	**85.78%**	**86.12%**	**86.04%**
	Loss	0.238	0.279	0.258	0.254

into two-dimensional matrices, which were used as input data in the convolutional neural network for training and modeling. From the coding in the two-dimensional matrices, the differences in strengths between players could also be observed.

5 Conclusions

This study used a total of four experiments to verify the feasibility of the proposed encoding method. The final results were compared with those of the traditional artificial neural network. We found the accuracy of the convolutional neural network to be slightly better than that of the artificial neural network. By taking advantage of the regional feature extraction capability, the convolutional neural network was able to achieve results that could be superior to that of the traditional artificial neural network. Moreover, the encoding method proposed in this research allowed the strengths of the players to be perceived via the matrices. This eliminated the need to analyze and compare the players separately in the original data.

This research has several limitations. First, in comparison with other kinds of ball sports, basketball has more controllable factors. Besides, its statistics are available

Table 11 Best results of Model 1 in each experiment

Batch size	Experimental result	Experiment 1	Experiment 2	Experiment 3	Experiment 4
32	Accuracy	90.74%	90.86%	73.21%	86.65%
	Loss	0.182	0.246	0.421	0.274
64	Accuracy	91.31%	91.12%	73.31%	86.71%
	Loss	0.177	0.211	0.451	0.193
128	Accuracy	91.43%	91.57%	73.59%	87.01%
	Loss	0.211	0.183	0.455	0.253
256	Accuracy	**91.78%**	91.62%	73.41%	87.44%
	Loss	0.207	0.205	0.457	0.255
512	Accuracy	91.24%	**91.67%**	**73.60%**	**87.48%**
	Loss	0.172	0.212	0.450	0.230

Table 12 The artificial neural network parameter settings and experimental results

	ANN	CNN
Hidden layer	1	1
Iteration	200	200
Batch size	256	256
Learning rate	0.0015	0.0015
Accuracy	91.64%	91.78%

to the public. Thus, this study selected basketball as the research topic. Second, the Python suite and programming language we selected for our deep learning development requires the Linux operating system in order to be fully supported. Third, the size of the matrix encoded in this research was limited to 16 × 16 because the NBA allows no more than 16 players to be registered at a time for a given game.

As mentioned in Sect. 3.2, the variables of the NBA data are under analysis in this research. The method in this research encodes the characteristics of basketball only. In the future, more sports events will be encoded for the modeling of the convolutional neural network. Adjustments of the model architecture can also raise the performance and increase the accuracy of the prediction results.

References

1. Google: Google Trend. https://trends.google.com.tw/trends/?geo=TW (2018). Accessed 4 June 2018
2. Zhao, W.X., Wu, H.H.: Japan's first–full of actuarial science. Business Weekly 1500 (2016)
3. Fong, R.S.: Studies on predicting the outcome of professional baseball games with data mining techniques: MLB as a case. Department of Information Management of Chinese Culture University. Unpublished Thesis (2013)
4. Hubel, D.H., Wiesel, T.N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. J. Physiol. **160**(1), 106–154 (1962)
5. Fukushima, K., Miyake, S.: Neocognitron: a self-organizing neural network model for a mechanism of visual pattern recognition. In: Competition and Cooperation in Neural Nets, pp. 267–285. Springer, Berlin (1982)
6. Indolia, S., Goswami, A.K., Mishra, S.P., Asopa, P.: Conceptual understanding of convolutional neural network-a deep learning approach. Procedia Comput. Sci. **132**, 679–688 (2018)
7. Craig, C., Overbeek, R.W., Condon, M.V., Rinaldo, S.B.: A relationship between temperature and aggression in NFL football penalties. J. Sport. Health. Sci. **5**(2), 205–210 (2016)
8. Maszczyk, A., Gołaś, A., Pietraszewski, P., Roczniok, R., Zając, A., Stanula, A.: Application of neural and regression models in sports results prediction. Procedia Soc. Behav. Sci. **117**, 482–487 (2014)
9. Bačić, B.: Towards the next generation of exergames: flexible and personalised assessment-based identification of tennis swings. In: 2018 International Joint Conference on Neural Networks (2018). https://doi.org/10.1109/ijcnn.2018.8489602
10. Bunker, R.P., Thabtah, F.: A machine learning framework for sport result prediction. Appl. Comput. Inform. **15**(1), 27–33 (2019)
11. Kipp, K., Giordanelli, M., Geiser, C.: Predicting net joint moments during a weightlifting exercise with a neural network model. J. Biomech. **74**, 225–229 (2018)
12. NBA Media Ventures: NBA official website. https://nba.udn.com/nba/index (2018). Accessed 4 June 2018
13. Manley, M.: Martin Manleys Basketball Heaven. Doubleday Books (1989)
14. Kline, D.M., Berardi, V.L.: Revisiting squared-error and cross-entropy functions for training neural network classifiers. Neural Comput. Appl. **14**(4), 310–318 (2005)

Heterogeneous Computing System for Deep Learning

Mihaela Maliţa, George Vlăduţ Popescu and Gheorghe M. Ştefan

Abstract Various forms of Deep Neural Network (DNN) architectures are used as Deep Learning tools for neural inspired computational systems. The computational power, the bandwidth and the energy requested by the current developments of the domain are very high. The solutions offered by the current architectural environment are far from being efficient. We propose a hybrid computational system for running efficiently the training and inference DNN algorithms. The system is more energy efficient compared with the current solutions, and achieves a higher actual performance per peak performance ratio. The accelerator part of our heterogeneous system is a programmable many-core system with a Map-Scan/Reductive only the cells where architecture. The chapter describes and evaluates the proposed accelerator for the main computational intensive components of a DNN: the fully connected layer, the convolution layer, the pooling layer, and the softmax layer.

Keywords Deep neural network · Parallel computing · Heterogeneous computing · Accelerators

1 Introduction

The mono-core computation can no longer keep up with the increasing demand of computational power requested by Deep Learning applications, making a multi- and many-core approach inevitable. At the same time, two kind of computations are segregated from the homogenous corp of the general purpose computing: the *complex*

M. Maliţa
Saint Anselm College, Manchester, NH, USA
e-mail: mmalita@anselm.edu

G. V. Popescu · G. M. Ştefan (✉)
Politehnica University of Bucharest, Bucharest, Romania
e-mail: gheorghe.stefan@upb.ro

G. V. Popescu
e-mail: georgevlad.popescu@yahoo.com

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_10

computation and the *intense computation*. The complex computation is defined by a code with the size, S, expressed as a number of lines, in the same magnitude order with its execution time, T, expressed as a number of clock cycles. The intense computation is characterized by $S \ll T$. To optimize the power consumption, the execution time, and the area of chips, a solution based on a general purpose mono-core must be substituted with a heterogeneous system. Whenever possible, the normal approach is to have for the complex computation a host engine, while for the intense computation an accelerator. The host engine could be a mono-core or a multi-core (multi means few) computational engine; however, the accelerator must be a many-core (many means no matter how big n) computational engine.

Our proposal for the accelerator part of the heterogeneous system includes a general purpose Map-Scan-Reduce Accelerator (MSRA) based on previous research [2, 13, 17], implementations [16], and investigated applications [1, 12, 14].

The main fallacy regarding the parallel computational systems currently used in Deep Learning applications is: the use of a gathering of consecrated processing cores, or of a specialized many-core engine, or of a specialized systolic array of circuits could be the solution for accelerating the DNN computation. Even if the use of an Intel's Many Integrated Core (MIC), or of an Nvidia's Graphic Processing Unit (GPU), or of a Google's Tensor Processing Unit (TPU) circuit, is ready to hand, the outcome will be inefficient because of various architectural incongruities. The architectural suitability that we propose allows to reduce $2-3\times$ the energy, and to increase approximately $\sim 3\times$ the *actual performance/peak performance* ratio.

In the following sections we describe the main computational requirements for DNN, the state-of-the-art hardware involved in Machine Learning applications, our proposed accelerator, and the implementation and evaluation of the main layers of a DNN.

2 The Computational Components of a DNN Involved in Deep Learning

The computational components of a DNN are presented in this section. Two correlated issues challenge the implementation of the applications involving DNN: (1) the data transfer between the computational engine and the main memory of the system (unfortunately, the ghost of the *von Neumann Bottleneck* is still haunting us), and (2) the specific computations associated with the different types of DNN. We begin by addressing the second issue. The first issue will be discussed when the specific library of functions are defined and used in subsequent Sects. 4.5 and 5.1.

Deep learning, as a branch of Machine Learning, uses various types of DNN. The most notorious are: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short Term Memory (LSTM), Deep Belief Network (DBN). The main computational layers to be accelerated in all the previous types of DNN are: the fully connected neural network layer, convolution layer, pooling layer and softmax layer. Simple vector operations must be also considered in order to articulate properly the layers just listed in order to obtain the more complex layers such as for RNN or LSTM.

2.1 Fully Connected Layers

A fully connected layer of n neurons receives an m-component input vector and sends out another n-component vector. The input vector, $\mathbf{x} = [x_1\ x_2\ \ldots\ x_m]$, is multiplied with a $m \times n$ matrix of weights and the result is submitted to a non-linear activation function.

Formally, the transfer function of a neuron is:

$$o = f(\sum_{i=1}^{m} w_i x_i) = f(net) \tag{1}$$

where f, the activation function, has various forms. For example:

- the sigmoid function of form:

$$f(y) = \frac{2}{1 + exp(-\lambda y)} - 1 \tag{2}$$

  where the parameter λ determines the steepness of the continuous function f; for a big value of λ the function f becomes: $f(y) = sgn(y)$
- ReLU (rectified linear unit) of form:

$$f(y) = max(0, y) \tag{3}$$

The neuron works as a combinational circuit performing the scalar product of the input vector $\mathbf{x}$ with the weight vector $\mathbf{w} = [w_1\ w_2\ \ldots\ w_m]$ followed by the application of the activation function. The activation function f, when it supposes a complex computation, is simply implemented using a look-up table (LUT).

A fully connected feed-forward NN is now a collection of n m-input neurons. Each neuron receives the same input vector $\mathbf{x}$ and is characterized by its own weight vector $\mathbf{w}_i$. The entire NN provides the output vector:

$$\mathbf{o} = [o_1\ o_2\ \ldots\ o_n] \tag{4}$$

The activation function is the same for each neuron. Thus, each NN is characterized by the weight matrix:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \dots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mn} \end{bmatrix} \tag{5}$$

with each neuron having its own column of weights. The computation consists of a matrix-vector multiplication followed by the application of the activation function on each component of the resulting vector.

2.2 Convolution Layer

Rather than using neurons to look at the entire input at a time, a convolution layer "scans" the input, crossing over the entire input with a small, $k \times k$, receptive field.

Let us consider the two-dimension input plan represented in the following matrix:

$$I = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{p1} & x_{p2} & \dots & x_{pp} \end{bmatrix} \tag{6}$$

where x_{ij} are scalars (to make the story short and simple we considered a square matrix). For example, I represents the 8-bit pixels of one of the RGB plans associated with a color image. The image will be scanned looking each time to a $k \times k$ *receptive field* of the following form:

$$R_{ij} = \begin{bmatrix} x_{ij} & x_{i(j+1)} & \dots & x_{i(j+k-1)} \\ x_{(i+1)j} & x_{(i+1)(j+1)} & \dots & x_{(i+1)(j+k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(i+k-1)j} & x_{(i+k-1)(j+1)} & \dots & x_{(i+k-1)(j+k-1)} \end{bmatrix} \tag{7}$$

Starting from the top left corner of the input plane, the first receptive field is R_{11}. Additional receptive fields are considered with a stride s horizontally and vertically:

$$i = 1, (1+s), (1+2s), \dots, (1+((p-k)/s)s) = 1, (1+s), (1+2s), \dots, (1+p-k)$$

$$j = 1, (1+s), (1+2s), \dots, (1+p-k)$$

where the stride could take values $s = 1, \dots, k$ (the stride cannot be bigger than k because the entire image must be scanned). If needed, the matrix I will be padded with zeroes to have $(p-k)/s = integer$.

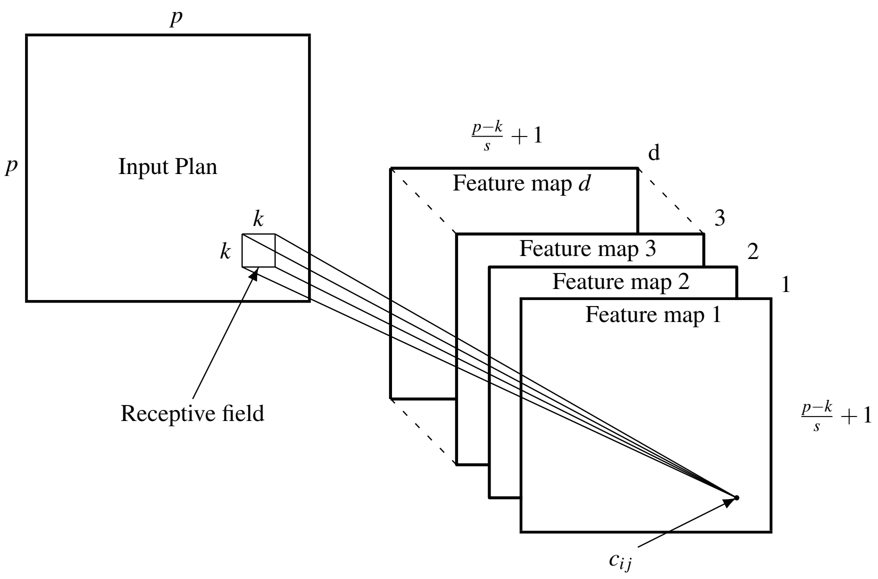


Fig. 1 Convolution

The neuron is the same during the scan of the entire input plan. This is called a *filter* and is defined as a matrix having the same size with the receptive field. For each input plane d filters are defined:

$$F^y = \begin{bmatrix} f^y_{11} & f^y_{12} & \cdots & f^y_{1k} \\ f^y_{21} & f^y_{22} & \cdots & f^y_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ f^y_{k1} & f^y_{k2} & \cdots & f^y_{kk} \end{bmatrix} \tag{8}$$

for $y = 1, 2, \ldots, d$. Each filter investigates the input plane “looking” for a specific feature, thus generating a *Feature map* (see Fig. 1). The filter F^y applied to the receptive field R_{ij} provides c^y_{ij} where:

$$c^y_{ij} = \sum_{m=1}^{k} \sum_{l=1}^{k} f^y_{lm} \times x_{(i+l-1)(j+m-1)} \tag{9}$$

Thus, the application of the filter F^y with stride s provides the matrix:

$$C^y = \begin{bmatrix} c^y_{11} & c^y_{12} & \cdots & c^y_{1(((p-k)/s)+1)} \\ c^y_{21} & c^y_{22} & \cdots & c^y_{2(((p-k)/s)+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c^y_{(((p-k)/s)+1)1} & c^y_{(((p-k)/s)+1)2} & \cdots & c^y_{(((p-k)/s)+1)(((p-k)/s)+1)} \end{bmatrix} \tag{10}$$

A convolutional layer consists of the application of d filters on the input plan generating a three dimensional array of $(((p-k)/s)+1) \times (((p-k)/s)+1) \times d$ scalars (see Fig. 1). For each filter a feature plan is generated with a scalar for every receptive field.

2.3 Pooling Layer

The pooling layer is used to reduce the size of a feature plan substituting (usually) a square *pooling window* of $q \times q$ scalars with only one scalar, which characterizes the entire pooling window. The scalar could be the maximum value from the pooling window, the sum of the values from the pooling window, or another value that is able to synthesize the content of the pooling window. The pooling windows are considered (usually) with a stride q in both directions in order to cover the entire feature plan. A stride smaller than q is possible, but it is not frequently considered.

Starting from a feature plan provided by a convolution, the pooling operation provides the pooled plan. Let us consider defining the pooling function with the same input I of $p \times p$ scalars. If the pooling window is $q \times q$ and the stride q (the usual case) the resulting plan is a $p/q \times p/q$ matrix of scalars P_q.

$$P_q = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1(p/q)} \\ y_{21} & y_{22} & \cdots & y_{2(p/q)} \\ \vdots & \vdots & \ddots & \vdots \\ y_{(p/q)1} & y_{(p/q)2} & \cdots & y_{(p/q)(p/q)} \end{bmatrix} \tag{11}$$

where y_{ij} is computed usually in two ways (see Fig. 3) for $q \times q$ matrices:

- by adding all the $q \times q$ values
- by taking the maximum value from the $q \times q$ values

Pooling window of $q \times q$ scalars in the matrix I results in a scalar in the P_q matrix (see Fig. 2)

In current applications, the value of q is 2 or 4. In Fig. 3 two examples for $q = 2$ are presented. One with Max as pooling function, and another with Add as pooling function. Each 2×2 matrix of scalars is substituted with their maximum or their sum.

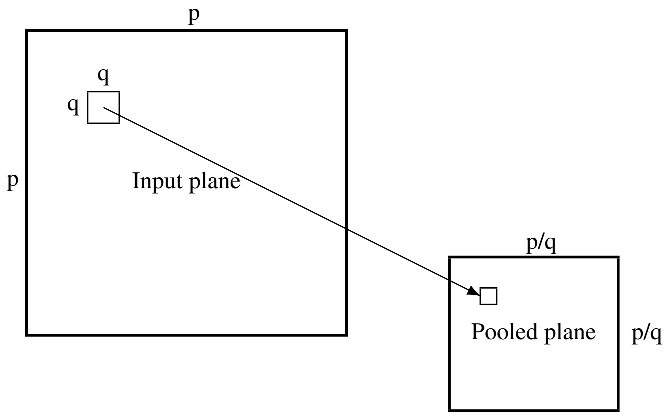
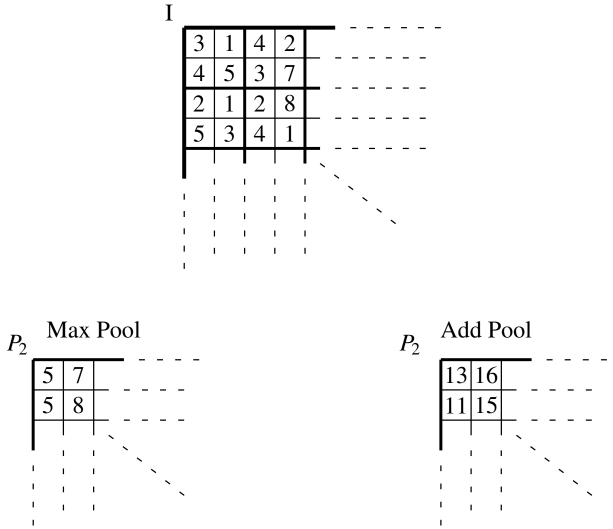


Fig. 2 The pooling operation: starting from a $p \times p$ matrix, results in a $p/q \times p/q$ matrix



I

3	1	4	2
4	5	3	7
2	1	2	8
5	3	4	1

Max Pool

5	7
5	8

Add Pool

13	16
11	15

Fig. 3 Examples of pooling for 2 × 2 pooling windows and stride 2. The Max Pool operation takes from the window the maximum value, while the Add Pool operation sums all the values from the window

2.4 Softmax Layer

The softmax layer is used for multi-category classification, in order to emphasis the most probable candidate as result. It is applied to a n-component vector $V = \langle x_1, x_2, \ldots, x_n \rangle$. Its value is determined by the standard exponential function on each component, divided by the sum of the exponential function applied to each component, as a normalizing constant. Therefore, the output components sum to 1:

$$\sigma_i(V) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \tag{12}$$

Results:

$$S_i(V) = \langle \sigma_1(V), \sigma_2(V), \ldots, \sigma_n(V) \rangle \tag{13}$$

In [18] the computation is simplified by avoiding the divide operation and by reducing the domain of the exponent. The first step is to down-scale the exponentiation:

$$\sigma_i(V) = \frac{e^{x_i}/e^{x_{max}}}{(\sum_{i=1}^{n} e^{x_i})/e^{x_{max}}} = \frac{e^{x_i - x_{max}}}{\sum_{i=1}^{n} e^{x_i - x_{max}}} \tag{14}$$

The second step is to compute the natural logarithm:

$$ln(\sigma_i(V)) = (x_i - x_{max}) - ln(\sum_{i=1}^{n} e^{x_i - x_{max}}) \tag{15}$$

While the sum in Expression (12) is susceptible to overflow because the values generated by exponentiation are high, and the divide operation is resource and time consuming, the Expression (14) works with smaller numbers and avoids the division. Both, *ln* and *exp* operations are performed using LUTs.

2.5 Putting All Together

An example of DNN is shown in Fig. 4, where all the previously presented functions are used to define a particular network. The input is a color image represented by the three color plans RGB. A first convolutional level with ReLU as activation function is followed by a pooling layer which is used to downsize the feature plans. We follow similar stages (convolution with ReLU and pooling) until the last pooled volume is flattened to a vector applied to a fully connected NN. The last stage is a softmax layer which provides the probabilities associated with possible input images.

While the first few convolutions are used to inspect the input to identify specific *local* features, the last fully connected layers provide a *global* analysis, and the softmax output layer emphasizes the most probable result.

3 The State of the Art

The main drawbacks of the current hardware solutions are: (1) the programmable (CPU, MIC, GPU, DSP) solutions are implemented on inappropriate parallel engines unable to use efficiently their high computational power in the specific computation requested by DNN, (2) the specific circuit solutions do not have enough flexibility in

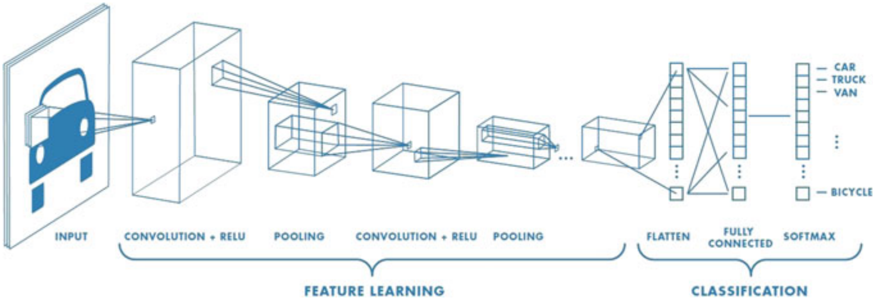


Fig. 4 An example of DNN [19]

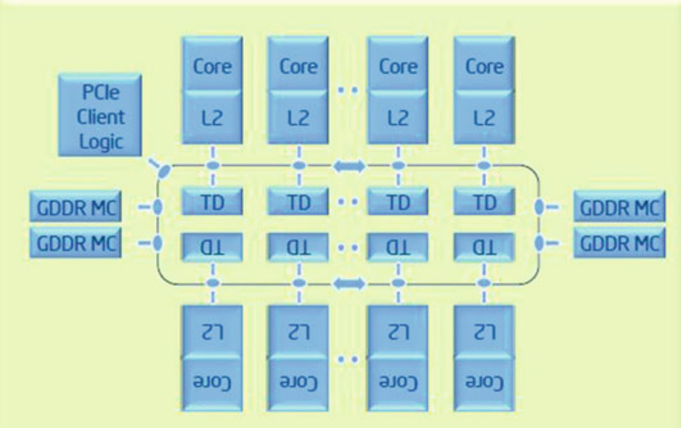


Fig. 5 Xeon Phi Micro-architecture with 2 levels of cache and ring architecture [21]

order to be efficiently adapted to the various forms of DNN, and (3) the FPGA-based reconfigurable solutions are too expensive, consume too much energy and require hardware specific knowledge. In the following subsections we will review the main solutions based on "of-the-shelf" computational devices.

3.1 Intel's MIC

Central Processing Units were not initially designed for machine learning, but in the last few years manufacturers began including multiple processing units that allow parallel execution of different tasks, making them a good candidate for deep learning. **Intel Xeon Phi** is the first product based on Intel's MIC Architecture (see Fig. 5).

Intel Xeon Platinum 8180 is a multi-core processor that offers a peak performance of 3259 GFLOPS for LINPACK [10] and 3.8 TFLOPS for SGEMM (Single precision floating General Matrix Multiply) using AVX512 [5]. The 28-core processor is fabricated using 14 nm technology and the TDP is 205 Watt, meaning a performance of 18.53 GFLOPS/W. This processor was evaluated in [5] for ResNet-50 topology and the obtained forward and backward propagation performance for the majority of the layers is 70–80% of the machine peak. Nevertheless, there are layers where the performance is about 55% of the peak, because of the high bandwidth requirements for the process of writing the output tensors. An important observation is that the previous performance results were obtained for an optimized implementation called direct convolutional kernels. For other convolution implementation, such as `im2col`, `libxsmm` or `autovec`, the performance is much lower: 3 times smaller for `im2col`, 9 times smaller for `libxsmm` and 16 times smaller for `autovec`.

Intel Xeon Phi 7295 is a processor specialized for deep learning, with 72 cores and a peak performance of 11.5 TFLOPS for SGEMM using FMA4 instruction set. The processor is fabricated using 14 nm technology and the TDP is 320 Watt, meaning the FLOP/Watt performance is 35.9 GFLOPS/W.

This type of processor was also evaluated in [5] and the performance varied according to the filter dimensions. For example, layers with 1×1 filters achieve $\sim$50% of their peak performance and layers with 3×3 filters achieve 70% of their peak. For other convolutional layer implementations, the performance is even smaller than the one obtained for Xeon Platinum 8180: a $\sim$20% of peak average for `im2col` and just a few percent of peak for most of the layers, when using `libxsmm` or `autovec`.

Intel Xeon E5-2699 v3 (Haswell) CPU with 18 cores and a peak performance of 1.3 TFLOPS for a TDP of 145 Watt (8.96 GFLOPS/W, 22 nm technology) was evaluated in [8] for six DNN applications: two MLP networks, containing fully connected layers, two LSTM networks, containing fully connected and element-wise operation layers) and two CNN networks, containing convolutional, pooling and fully connected layers. The performance evaluation reveals that the CNN networks use 23 and 46% of processor peak computation capabilities, the MLP networks use 15.4 and 38.5% of processor peak computation capabilities and LSTM networks use 84.6 and 46% of processor peak computation capabilities.

3.2 Nvidia's GPU as GPGPU

GPUs are processors created for computer graphics (see Fig. 6), which, due to their high number of cores and high parallelism, are very effective in running matrix multiplications, the main operation involved in deep learning.

The most famous GPU manufacturer is NVIDIA. Besides the usual GPUs, during the last years they have begun to make their products more efficient in artificial intelligence tasks.

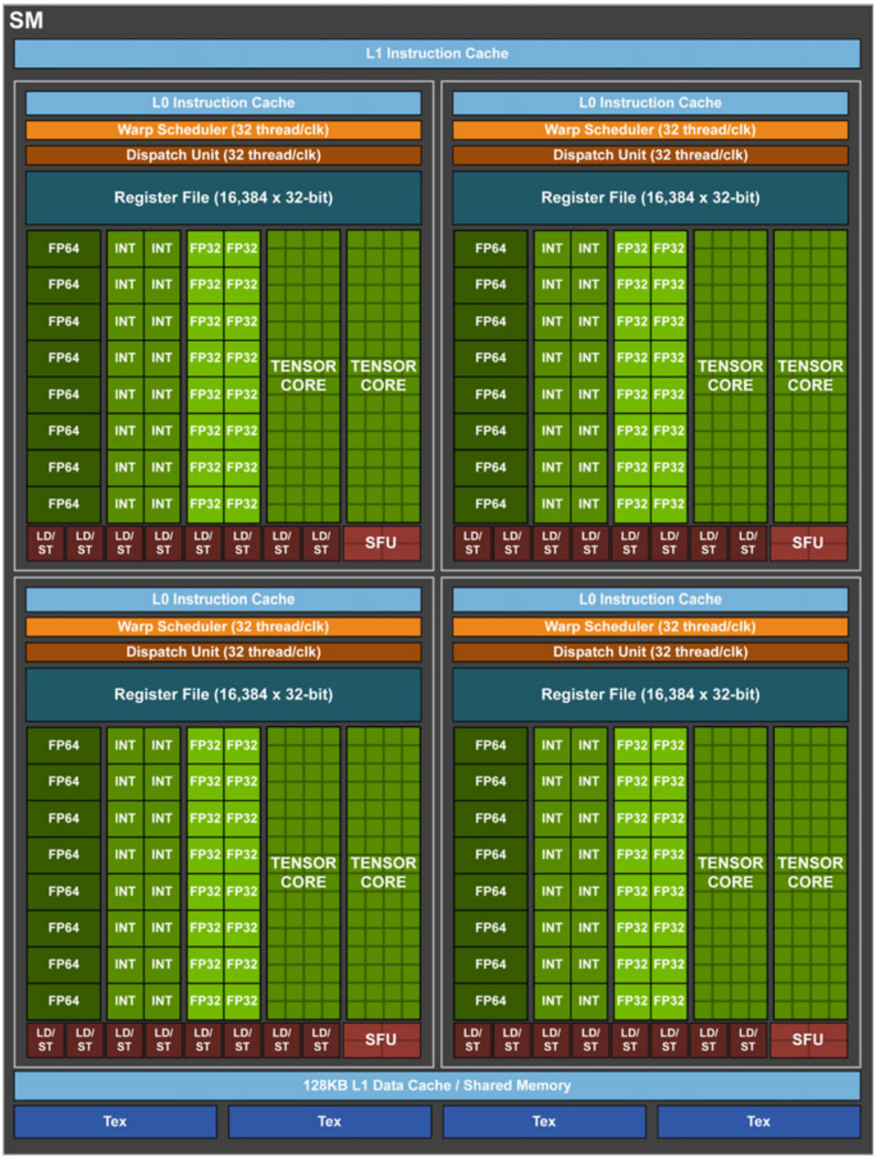

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Fig. 6 NVIDIA Titan V [15]

Although the GPUs have large computational capabilities, the real performance obtained in deep learning applications may be far from their peak performance. The factors that can influence the efficient use of computational and power resources are either related to processor architecture or software that is not optimal for the architecture it targets. This gap between theoretical and real performance for GPUs has been highlighted in several papers.

NVIDIA GTX Titan Black is a GPU with 2880 CUDA cores and 5645 GFLOPS single precision floating point (FP32) peak performance. The GPU was fabricated in 28 nm technology and the TDP is 250 W, which means a FLOP/Watt performance of 22.56 GFLOPS/W. The use of computational performance on CNN experiments are positioned in the range of 9–50% from peak [11].

NVIDIA Tesla K40 is a GPU with 2880 CUDA cores and 4.29 TFLOPS single precision peak performance (28 nm technology, TDP = 235 W, 18.25 GFLOPS/W). Different convolutional layers were tested and the obtained performance ranges from 23 to 35% of peak [3].

NVIDIA Geforce GTX 980 is a GPU with 2048 CUDA cores and a peak single precision performance of 4.95 TFLOPS (28 nm technology, TDP = 165 W, 30 GFLOPS/W). Different convolutional layers were tested and the obtained performance ranges from 30 to 51% for NVIDIA Geforce GTX 980 [3].

NVIDIA Tesla K80 is a card containing two GPUs, each one with 2496 CUDA cores and a peak single precision floating point performance of 2.8 TFLOPS (28 nm technology, TDP = 150 W, 18.66 GFLOPS/W). The evaluation for two MLP networks, two LSTM networks and two CNN networks reveals that the CNN networks use 32.1 and 35.7% of peak performance, the MLP networks use 7.14 and 25% of peak performance and LSTM networks use 17.85 and 25% of peak performance [8].

Although many applications require high precision computation (32-bit floating point FP32, or 64-bit floating point FP64), researchers have discovered that a half precision floating point (FP16) is sufficient for deep learning training. Additionally, deep learning inference can be performed using 8-bit integer computation, without significant impact on accuracy [6]. In order to make GPUs more efficient in performing different tasks, multiple precision modes are supported.

Starting with the Volta generation, a specialized Tensor Core unit was added, speeding up the matrix multiplications. Volta Tensor Cores combines FP16 (half floating point precision) multiplications with FP32 accumulations. The newer Turing Tensor Cores are enhanced for inferencing, adding new INT8 and INT4 precision modes. In order to evaluate the performance of GPUs using those specialized cores, a new performance metric was defined: Tensor Tera Operations Per Second, TTOPS.

Example of GPUs optimized for deep leaning are NVIDIA Titan V (see Fig. 6) and NVIDIA Tesla V100. Titan V contains 5120 CUDA Cores and 640 Tensor Cores and offers a performance of 13.8 TFLOPS for single precision floating-point (FP32), 27.6 TFLOPS for half precision floating-point (FP16) and 110 TTOPS for deep learning. Tesla V100 (PCIe) contains 5120 CUDA Cores and 640 Tensor Cores and offers a

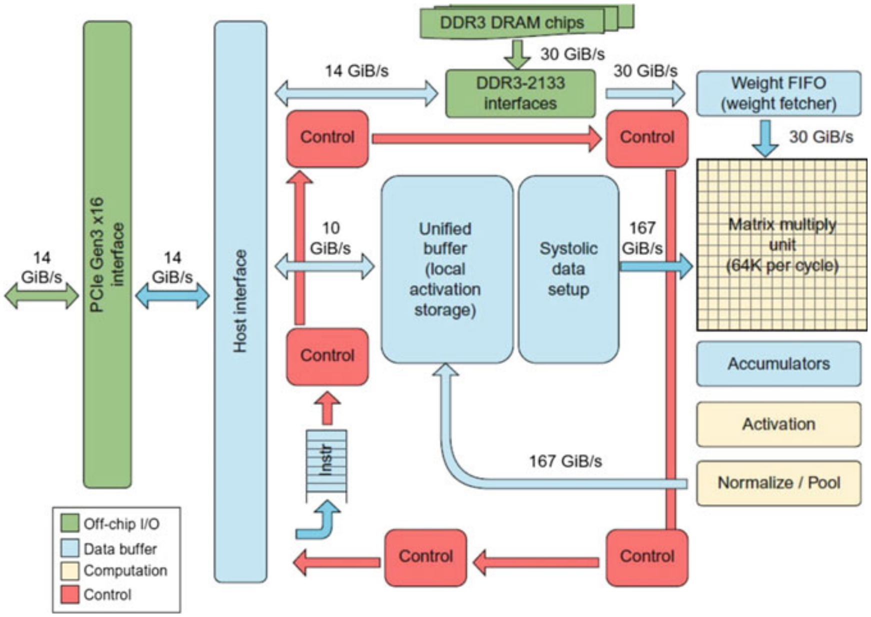


Fig. 7 Block diagram of Tensor Processing Unit (TPU) [8]

performance of 14 TFLOPS for single precision floating-point (FP32), 28 TFLOPS for half precision floating-point (FP16) and 112 TTOPS for deep learning.

3.3 Google's TPUs

TPU is an Application-Specific Integrated Circuit (ASIC) for neural networks inference, used as accelerator in a hybrid system, the communication between TPU and host being assured by a PCIe bus.

The core of the chip (see Fig. 7) is a systolic array of 256 × 256 8-bit multipliers, called Matrix Multiply Unit (MXU), which performs matrix multiplications between input data and weights. The MXU input data is stored in the Unified Buffer, which helds the results of previous computation steps. The data transfer between Unified Buffer and the host memory is controlled by a DMA controller. The MXU input weights are delivered by the Weights FIFO.

The resulting products are accumulated in the Accumulators and the nonlinear activation functions are computed by the Activation Unit.

Running at a frequency of 700 MHz and computing 256 × 256 multiply-and-adds for 8-bit integers every clock cycles, the peak performance of TPU is 92 TTOPS. The chip was fabricated in 28 nm technology and the TDP is 75 W, meaning that the TTOPS/Watt performance is 1.22 TTOPS/W.

TPU was evaluated in [8] for the same six DNN applications described above. The MLP networks use 13.3 and 10.5% of its peak computing capabilities, LSTM networks use 4 and 3% of its peak computing capabilities and CNN networks use 93.4 and 15.3% of its peak computing capabilities, the mean performance being 23.24% of its peak. The performance for MLPs and LSTMs is limited by memory bandwidth, while the small performance of one of the CNNs networks can be explained by its structure (the presence of fully connected layers and the shallow feature depth of some layers).

Google also developed TPU v2 and TPU v3. If the initial TPU was limited to 8-bit integer operations, the new generations can also calculate in floating point (the MXU units perform multiplies at reduce `bfloat16` precision [20]), allowing it to be used also for neural networks training.

The second generation of TPU has two 128 × 128 MXUs, each one connected to an 8 GB High Bandwidth Memory, increasing the bandwidth to 600 GB/s. The peak performance for each TPU v2 chip is 45 TTOPS [9].

The third generation of TPU has two cores, each one with two 128 × 128 MXU units, peak performance being twice as the previous generation one.

3.4 Concluding About the State of the Art

Putting together all the information from the previous analysis (see Table 1), we can learn some very important things about the way to define the structure and the architecture of an accelerator.

The general purpose architectures implemented by MICs provide a pretty good actual performance from the peak performance (an average of $\simeq$45%) but the computation per Watt is relatively low (an average of $\simeq$21 GFLOP/W). The reduced number of cores (less than 100) requests a simple control, allowing, in some applications for optimized code, the use of $>$80% from the peak performance. But, most of the applications, evaluated for general purpose multi-cores, are unable to use more than 25% from their peak performance, and some of them use only few percentage of their very high performance. The architecture of the general purpose computers are designed for a wide specter of applications, while for intense applications there are specific requirements. The general purpose processors waste too much resources for 32-bit floating point computations, while usually the CNN computation asks small integer arithmetic for inferences and accepts 16-bit float operations for training.

General purpose graphic processing unit (GPGPU) is an oxymoron. It is tempting to take "of-the-shelf" a many-core parallel processor to solve intense computations, but, in the same time, is dangerous to use a powerful processor far from its application domain. A graphic machine can not be converted in a machine learning accelerator without a high risk. The actual performance related to the peak performance is lower compared with general purpose multi-core architectures (average $\simeq$34%) due to the difficulties involved in control and data transfer for hundred or thousands of execution

Table 1 Overview for analyzed devices performance

Chip model	Fab. techn. (nm)	Peak performance	Performance per Watt	Actual % from peak
Intel Xeon Platinum 8180	14	3.8 TFLOPS	18.53 GFLOPS/W	5–80
Intel Xeon E5-2699 v3	22	1.3 TFLOPS	8.96 GFLOPS/W	15–84
Intel Xeon Phi 7295	14	11.5 TFLOPS	35.9 GFLOPS/W	20–70
NVIDIA GTX Titan Black	28	5.64 TFLOPS	22.56 GFLOPS/W	9–50
NVIDIA Tesla K40	28	4.29 TFLOPS	18.25 GFLOPS/W	23–35
NVIDIA Geforce GTX 980	28	4.95 TFLOPS	30 GFLOPS/W	30–51
NVIDIA Tesla K80	28	2.8 TFLOPS	18.66 GFLOPS/W	7–35
Tensor Processing Unit	28	92 TTOPS	1.22 TTOPS/W	3–93

units. The energy use is only a little improved (an average of $\simeq$22.36 GFLOP/W). The GPUs optimized for deep learning are designed with distinct physical resources for integer, 32-float, 64-float, and tensor operations (see Fig. 6). Thus the area efficiency is lowered because too many times big parts of the area of the chip is unused. The cache approach persists with its inefficiency in helping the intense computation which is highly predictable requesting only a buffer-centered approach in the memory hierarchy design.

Specific ASIC's, such as TPU, do not have enough flexibility to support the high variety of DNNs. Therefore, only for a small part of the applications their huge computational power can be activated. For most of the applications (90%, according to [7]) no more than 13.4% from the peak performance is used. Only for one application, deployed in less than 5% of the applications, 93% from the peak performance is activated. The very big number of arithmetic systolic units (multipliers and adders) can not be easy put to work efficiently for the high variety of DNN we are facing in real applications.

The cache-based memory hierarchy is inappropriate for intense computation because of the high predictability of the program and data flow.

Architectural inadequacy is the main issue A general purpose architecture or a graphic architecture or a simple systolic circuit are hard to be adapted to the specific requirements of the intense computational domain of machine learning. And when the energy saving criteria is added, the problem becomes much harder.

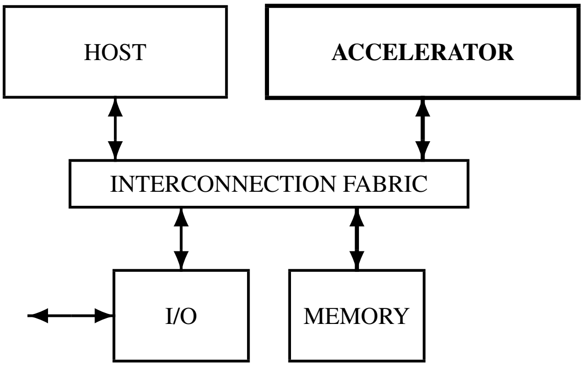


Fig. 8 Heterogeneous system

4 Map-Scan/Reduce Accelerator

4.1 The Heterogeneous System

The computation becomes “hybrid” or heterogeneous when we start to segregate the execution of a program in two tightly interleaved parts:

- intense computations, characterized by short code and big execution time; these parts are sent to an accelerator
- complex computations, when the size of code and the execution time are in the same magnitude order; these parts run on the host.

Thus, the pair HOST & ACCELERATOR represents the structure of a HETEROGENEOUS COMPUTER. A possible embodiment is presented in Fig. 8, where:

- HOST is a general purpose processor which runs the application using a library of functions written using a kernel library running on ACCELERATOR (for example the Eigen library or the TensorFlow library implemented using EigenKernel or TensorFlowKernel libraries)
- ACCELERATOR is our MSRA running EigenKernel or TensorFlowKernel libraries, i.e., Eigen or TensorFlow libraries limited to data structures managed efficiently in a n-cell accelerator
- INTERCONNECTION FABRIC is a multiple point interconnection network used for fast data transfer between the components of the system
- MEMORY stores the programs and the data to be processed
- I/O system to connect the heterogeneous computer to the external world.

The host processor is programmed in a high level language (C, C++, Python,...) while the accelerator’s kernel library is developed, in this early stage of the project, in assembly language in order to achieve the highest possible performance. From the kernel library to the targeted library the implementation is done in a high level language.

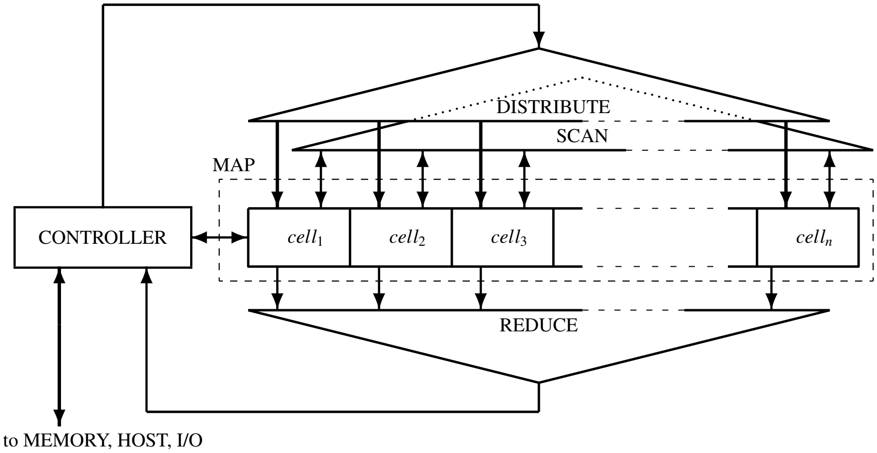


Fig. 9 Map-Scan/Reduce Accelerator (MSRA): the linear array of cells MAP with two global loops (one through REDUCE and another through SCAN) running code issued in each clock cycle by the CONTROLLER unit

4.2 The Accelerator's Structure

The structure of MSRA is presented in the current subsection. It is a general purpose programmable parallel accelerator optimized for the functional requirements described in Sect. 2. The accelerator is designed as part of a heterogeneous computing system in which the complex part of the program runs on the host computer, while the intense part of the application (convolution, pooling, fully connected NN) runs on the accelerator. The architecture and the structure of the heterogeneous system are described with emphasis on advantages provided for the investigated application domain.

MSRA is a n-cell engine (see MAP section in Fig. 9) with two global loops:

- one, closed directly through a log-depth scan circuit, SCAN, which receives a n-component vector from the array of cells and sends back a n-component vector
- another, closed through a log-depth reduction circuit, REDUCE, which receives a n-component vector from the array of cells and sends its output to CONTROLLER which issues in each cycle, through the log-depth network DISTRIBUTE, an instruction to be executed in each active cell.

The architectural image of MSRA for the user is:

- the constant vector index, distributed along the cells: $IX = [1, \ldots, n]$ used to identify the cells as $cell_1, cell_2, \ldots, cell_n$
- the distributed memory:

$$DM = \begin{bmatrix} s_{11} & s_{12} & \dots & s_{1n} \\ s_{21} & s_{22} & \dots & s_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & \dots & s_{mn} \end{bmatrix} \tag{16}$$

which can be seen as composed by

– full *horizontal vectors*, distributed along the cells, for $i = 1, \dots, m$:

$$H_i = \begin{bmatrix} s_{i1} \; s_{i2} \; \dots \; s_{in} \end{bmatrix} \tag{17}$$

whose components can be processed in parallel in the MAP section of the accelerator

– full *vertical vectors*, for $j = 1, \dots n$, each stored in the corresponding cell:

$$V_j = \begin{bmatrix} s_{1j} \\ s_{2j} \\ \vdots \\ s_{mj} \end{bmatrix} \tag{18}$$

whose components are stored in the m-location register file in each cell

- the Boolean vector distributed along the cells: $B = [b_1, \dots, b_n]$ used to activate the cells when $b_i = 1$
- the accumulator vector $ACC = [acc_1 \, acc_2 \dots acc_n]$ distributed along the cells
- CONTROLLER's data memory: $M = [s_1 \, s_2 \dots s_u]$
- the accumulator of CONTROLLER: acc

In the program memory of CONTROLLER, HOST loads programs whose binary form are stored as a pair of instructions, one for CONTROLLER and another to be executed in each active cell of the MAP array. Thus, in each clock cycle, from its program memory, CONTROLLER fetches an instruction for itself and another to be issued toward the MAP array. With a latency in $O(log\ n)$, CONTROLLER receives the result provided by the REDUCE network.

4.3 *The Micro-architecture*

There are the following types of operations performed by MSRA:

- data transfer operations
- spatial control operations targeting the content of the vector B whose components are used to perform predicated executions
- unary and binary predicated vector operations on horizontal vectors

- reduction operations on the components of the horizontal vectors provided by the active cells
- scan operations on the components of the horizontal vectors provided by the active cells.

4.3.1 Data Transfer Operations

The content of the distributed memory DM (Eq. 16) can be partially or totally loaded from or stored to MEMORY (see Fig. 8). The transfer is done one vector at a time using two functions:

- `load(s, i, j)`: s left most positions of the horizontal vector H_i are loaded with scalars from MEMORY starting at the address j
- `store(s, i, j)`: s left most scalars of the horizontal vector H_i are stored in MEMORY starting at the address j

Important note: the transfer between DM and MEMORY is transparent to the computation performed in our accelerator, i.e., during the transfer the computation runs in parallel undisturbed. This important feature of our architecture contributes to avoiding, at leas partially, the "bottleneck" between the MAP array and MEMORY. Smartly used, it feeds up the data hungry MAP array with data from MEMORY.

4.3.2 Spatial Control Operations

The spatial control allows the predicated execution by working on the value of the Boolean vector B. Each of the following operations is performed in one clock cycle:

- `activate`:
$$B \Leftarrow [1, 1, ..., 1]$$
activates all the cells of the accelerator
- `where(cond)`:
$$b_i = ((b_i == 1)\ \&\ cond_i)\ ?\ 1\ :\ 0$$
in all active cells, keeps active only the cells where the condition $cond_i$ is fulfilled, where $cond_i$ stands for the value the condition `cond` takes in $cell_i$
- `elsewhere`: in all the cells active before the action of the last recent still lasting `where(...)`, keeps active only the cells where the condition $cond_i$ is not fulfilled, i.e., performs the complementary `where` selection in the active space
- `endwhere`: restores the vector B to the state before the last recent still lasting `where(...)`

Embedded `where(...)` are allowed.

Example 4.1 Let be, for $n = 16$ the boolean vector uninitiated and the accumulator vector loaded with the index vector IX:

```
B = [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-]
ACC = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

The evolution of the Boolean vector B under a sequence of spatial control operations is the following:

```
(1) activate;          => B = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(2) where(acc > 3);    => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3) where(acc < 14);   => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0]
(4) elsewhere;         => B = [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1]
(5) endwhere;          => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1]
(6) endwhere;          => B = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

In step (1) all the cells are activated: $b_i = 1$, for $i = 1, 2, \ldots, 16$. Then, remain active only the cells where $acc_i > 3$. In the next step, the embedded `where` operation is exemplified: from the active cells remain active only the cells where $acc_i < 14$. The `elsewhere` operation is performed in the active space selected by `WHERE(ACC > 3)`. Step (4) activates the cells inactivated by the previous `where(...)`. Step (5) restores the vector B to the state before the last recent `where(...)` where action still last: `WHERE(ACC < 14)`. In step (6), vector B is restored to its initial value stated by the operation `ACTIVATE`. ◇

4.3.3 Arithmetic and Logic Operations

Each vector operation is performed on n-component horizontal vectors in constant time.

Unary Operations. Some operations are performed only in the active cells where the value of the Boolean vector `B` is bi $=$ `1`, other are performed in all cells, independent of the content of `B`. For example:

- `inc(i, k)`:

  $$s_{ij} \Leftarrow b_j \; ? \; (s_{kj} + 1) \; : \; s_{ij}$$

  for $j = 1, \ldots, n$; in each active cells, the component of the horizontal vector H_i takes the incremented component of the horizontal vector H_k
- `shiftRight(i, k, input)`:

  $$s_{ij} \Leftarrow (j == 0) \; ? \; input \; : \; s_{k(j-1)}$$

  for $j = 1, \ldots, n$; all the components of the horizontal vector H_k are shifted one position right with $input$ inserted in the left end position, and are stored as H_i

Binary Operations. The binary operations are performed in constant time (usually one clock cycle) only in the active cells where the value of the Boolean vector `B` is 1. For example:

- `mult(k, l, m)`:
  $$s_{kj} \Leftarrow b_j \ ?\ (s_{lj} * s_{mj}) \ :\ s_{kj}$$
  for $j = 1, \ldots, n$; in each active cells, the component of the horizontal vector H_i takes the value of the product between the corresponding components of the horizontal vectors H_l and H_m
- `xor(k, l, m)`:
  $$s_{kj} \Leftarrow b_j \ ?\ (s_{lj} \oplus s_{mj}) \ :\ s_{kj}$$
  for $j = 1, \ldots, n$; in each active cells, the component of the horizontal vector H_i takes the value of the bitwise XOR between the corresponding components of the horizontal vectors H_l and H_m

4.3.4 Reduction Operations

The reduction operations are performed on the values provided by the active cells. They are:

- `redadd(i)`:
  $$acc \Leftarrow \sum_{k=1}^{n} (b_k \ ?\ s_{ik} : 0)$$
  the CONTROLLER's accumulator takes the sum of the accumulators of the active cells with a latency $\mathscr{L} \in O(log\ n)$
- `redmax(i)`:
  $$acc \Leftarrow Max_{k=1}^{n} (b_k \ ?\ s_{ik} : 0)$$
  the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the maximum value stored in the accumulators of the active cells
- `redmin(i)`:
  $$acc \Leftarrow Min_{k=1}^{n} (b_k \ ?\ s_{ik} : \infty)$$
  the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the minimum value stored in the accumulators of the active cells; the symbol ∞ stands for the biggest number in the representation used in the application.
- `redbool`:
  $$acc \Leftarrow OR_{k=1}^{n} b_k$$
  the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the logic OR from all the bits of the Boolean vector B (used to test if at least one cell is active or not).

4.3.5 Scan Operations

The scan operations take a vector, HV_k, from LM and return a vector, HV_i, whose components are computed according to the global content of HV_k. For example:

- `prefixadd(i, k)`: $HV_i \Leftarrow [(b_1?s_{k1} : 0), (\sum_{j=1}^{2}(b_j?s_{kj} : 0)), \ldots, (\sum_{j=1}^{n}(b_j?s_{kj} : 0))]$
- `compact(i, k)`: $HV_i \Leftarrow [s_{k1}, s_{k3}, \ldots, s_{k(n-3)}, s_{k(n-1)}, \underbrace{0, 0, \ldots, 0}_{n/2}]$ aligns to the left the odd components of the vector.

4.4 Hardware Parameters of MSRA

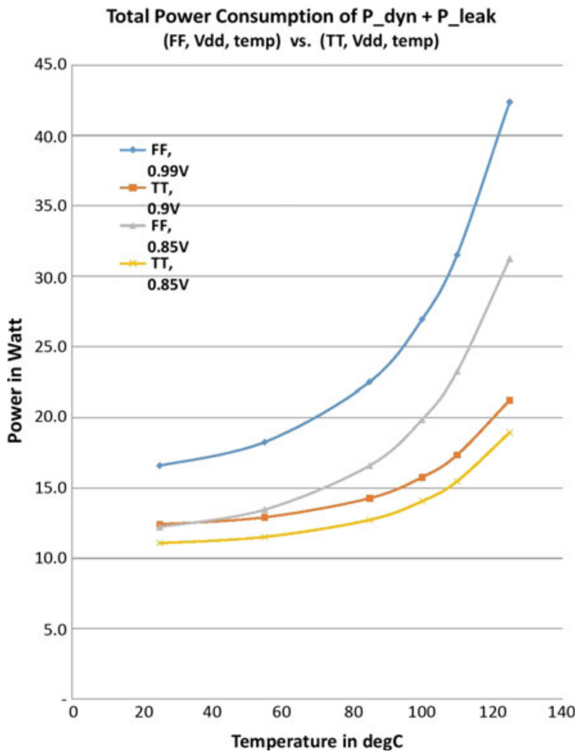
The hardware performances are evaluated using simulation tools for the 28 nm technology. The simulation of a MSRA running at 1 GHz was used to evaluate the size and the power for a version having a 32-bit DDR interface, the 32-bit word size, the number of cells $n = 2048$, and the memory size $m = 1024$ (4 KB SRAM/cell). The resulting area of the chip is $9.2 \times 9.2\ \text{mm}^2 = 84.64\ \text{mm}^2$. The power consumed by the chip is shown in Fig. 10. Depending on temperature results: 12/14/18 W at 80/100/120 °C. The computational performances are:

- for integer arithmetic:
  - 2048 GOPS, where GOPS stands for Giga 32-bit Operations Per Second; at 80 °C results 170.66 GOPS/W
  - 4096 GOPS, for 16-bit operations; at 80 °C results 341.33 GOPS/W
- for applications involving floating point arithmetic with 20% float operations plus 80% integer operations:
  - 820 GOPS for float and integer operations defined on 32 bits; at 80 °C results 68.33 GOPS/W
  - 2400 GOPS for float and integer operations defined on 16 bits; at 80 °C results 200.33 GOPS/W

  because the float operations are performed in a sequence of operations in execution units with no floating point units implemented as distinct structures, like in NVIDIA Titan V (see Fig. 6)
- for machine learning operations: 8 TTOPS, where TTOPS stands for Tensor TOPS (see Sect. 3.2); at 80 °C results 1.36 TTOPS/W, if the architecture is designed for tensor operations used in Machine Learning applications.[1]

[1] Tensor TOPS are evaluated (as for TPU) considering the REDUCE section performing adds in parallel with the MAP section which performs multiplications.

Fig. 10 Power consumption evaluated for out MSA [14]



To note that for training DNN the proposed architecture provides 200.33 GOPS/W, while for inference 341.33 GOPS/W or 0.68 TTOPS/W (around half of the circuit performance).

4.5 NeuralKernel *library*

For our application, a small library, let us call it *Neural*, can be implemented using the *NeuralKernel* library defined starting with the following functions:

`instmx(M, l, c)`: instantiate in ACCELERATOR's MAP array the matrix `M` with l lines and c columns, where the lines are parts of the full horizontal vectors (see Eq. 17), and columns are parts of the full vertical vectors (Eq. 18)

`instvt(V, l)`: instantiate in the data memory of ACCELERATOR's CONTROLLER the vector `V` of length l

`loadmx(M, p)`: load the content of the matrix `M` from MEMORY starting from the address `p` where the matrix is stored line by line, and increment the pointer `p` to $(\mathtt{p} + \mathtt{l} \times \mathtt{c})$

`storemx(M, p)`: store the content of the matrix `M` to MEMORY starting from the address `p` where the matrix is stored line by line, and increment the pointer `p` to $(\mathtt{p}+\mathtt{l}\times\mathtt{c})$

`loadvt(V, p)`: load the content of the vector `V` from MEMORY starting from the address `p`, and increment the pointer `p` to `(p + 1)`

`storevt(V, p)`: store the content of the vector `V` to MEMORY starting from the address `p`, and increment the pointer `p` to `(p + 1)`

`mmm(M1, M2, M3)`: the matrix `M1`, is multiplied with the matrix `M2` and the result is stored as `M3`

`conv(M1, V, M2, k, s)`: the matrix `M1` is convoluted, with stride `s`, using a $k \times k$ filter stored line by line in the vector `V` and the resulting feature plan is stored as `M2`

`sigmoid(M1, M2)`: the activation function sigmoid is applied to the matrix `M1` with result as `M2`

`relu(M1, M2)`: the activation function ReLU is applied to the matrix `M1` with result as `M2`

`pooladd(M1, M2, q, s)`: the content of the matrix `M1` is pooled to the sum of the values of the $q \times q$ pooling window with stride `s`, and the result is stored as matrix `M2`

`poolmax(M1, M2, q, s)`: the content of the matrix `M1` is pooled to the maximum of the values of the $q \times q$ pooling window with stride `s`, and the result is stored as matrix `M2`

`softmax(M1, M2)`: the softmax function is applied to `M1` with result as `M2`

where the size of the matrices and vectors are limited by the parameters n, m, u previously defined.

5 Implementation and Evaluation

Implementation and evaluation of the proposed system in implementing DNN are presented in this section. Roughly speaking, we use ***map*** resource to accelerate the convolution, the ***scan*** resource to accelerate the pooling, and the ***map-reduce*** resources for the fully connected NN. The algorithms we propose for the main computational patterns used in implementing a DNN are presented in versions easy to accelerate on our proposed architecture.

All the experiments we have done with the proposed architecture are performed on a cycle accurate simulator. The power estimate is based on the evaluation of the optimised synthesizable RTL. Therefore, in this stage of the project only the main functions of the *NeuralKernel* library are implemented and quantitatively evaluated.

5.1 Fully Connected NN

The fully connected layer of a NN consists of a matrix-vector multiplication, the application of an activation function to the resulting vector, and the associated data transfer process. The algorithm is described in Fig. 11. Three processes are running in parallel: (1) the control process, (2) the computation and (3) the data transfer.

The execution time is dominated, for small v by the load of the weight matrix (`load(M, p1)`), or, for big v by the loop **doInParallel**. The application must be designed, if possible, so as to maximize the value of v. Thus, the matrix `M` is loaded only once for many uses.

For big v the execution time is dominated by the slowest process executed in parallel in the **doInParallel** loop. Because the transfer time is executed in time belonging to $O(c)$, we pay attention to the main computational process: `{mmm(M, V, W); relu(W, W);}`. In Fig. 12, the algorithm for matrix-vector multiplica-

```
/*****************************************************************
FUNCTION NAME: Fully Connected Neural Network
    - Load the weight matrix M starting with mxPointer
    - Multiply M with 'v' vectors stored successively starting with
        the address pointed by inVectPointer
    - Apply the activation function ReLU
    - Store the resulting 'v' vectors successively starting with the
        address pointed by outVectPointer
*****************************************************************/
instmx(M,line,column);      // define the size of the weight matrix M
instmx(V,1,column);         // define the size of the input vector V
instmx(W,1,column);         // define the size of the output vector W
inits(p1,mxPointer)         // initiate the value of matrix pointer
inits(p2,inVectPointer);    // initiate the input vector pointer
inits(p3,outVectPointer);   // initiate the output vector pointer

fullyConnectedNN(M,p1,p2,p3,n);
    loadmx(M,p1); // load in ACCELERATOR the content of M from p1
    loadmx(V,p2); // load in ACCELERATOR the first V starting from p2
    mmm(M,V,W);   // W <= M x V
    relu(W,W);    // W <= ReLU(W)
    load(V,p2);   // load the next V starting from p2 + c
    i <= 0;
    doInParallel{
        {   while (i < v-1)
            i = i+1;          } // control process runs on CONTROLLER
        {   mmm(M,V,W);
            relu(W,W);        } // computation running in MAP+REDUCE
        {   storemx(W,p3);
            loadmx(V,p2);     } // transferring process
    }
    storemx(W,p3);              // the last transfer of the result W
```

Fig. 11 The algorithm for the fully connected neural network

```
/*****************************************************************
FUNCTION NAME: Matrix-vector multiplication: W <= M * V
    - the matrix M with the lines M[i], for i = 1,2, ... l
*****************************************************************/
    for(i=1; i=<l; i=i+1)
        shiftRight(S,S,redadd(mult(V,M[i],V));
    for(i=1; i<(log_2 n); i=i+1)
        no operation; // for the latency of the reduction add
    W <= S
```

Fig. 12 Matrix-vector multiplication

tion is presented. For `shiftRight(S, S, redadd(mult(V, M[i], V))` see Sect. 4.3 where the multiplication, shift and reduction operations are defined. The execution time for this operation is $2 + log_2 n$ when it is executed only once. For l executions, because of the pipelined hardware involved, the total execution time is $2 \times l + log_2 n$, where n is the number of cells in the MAP section of the ACCELERATOR. Thus, multiplying a $l \times c$ matrix with a c-component vector, with $c \leq n$, is executed in time belonging to $O(l)$, i.e., a n-cell ACCELERATOR provides an acceleration belonging to $O(n)$.

What is the constant associated to the acceleration in $O(n)$? The constant is for sure > 1, because in a mono-core processor the data transfer, the arithmetic operations and the control operation are performed sequentially, while in our architecture the **doInParallel** loop is possible because specific hardware is provided for the three processes. We can claim that, for this function, the acceleration is supra-linear.

5.2 Convolutional Layer

5.2.1 Stride $s = 1$

Step 1: load from the MEMORY the matrix I which is considered the input plan (Eq. 6), as p lines (vectors) each of p components, and the filters as d sets of $k \times k$ scalars, then set $y \Leftarrow 1$, and $b \Leftarrow 1$. The execution time $t_1 = const \times (p^2 + d \times k^2)$.

Step 2: we consider the filter F^f as k "vertical" vectors:

$$F_i^f = \begin{bmatrix} f_{1i}^f \\ \vdots \\ f_{ki}^f \end{bmatrix}$$

for $i = 1, \ldots, k$, and the p "vertical" vectors:

$$I_j^b = \begin{bmatrix} x_{bj} \\ \vdots \\ x_{(b+k-1)j} \end{bmatrix}$$

for $j = 1, \ldots, p$, associated to the “band” b. Then we compute the matrix:

$$\begin{bmatrix} I_1^b \times F_1^1 & I_2^b \times F_1^1 & \ldots & I_p^b \times F_1^1 \\ I_1^b \times F_2^1 & I_2^b \times F_2^1 & \ldots & I_p^b \times F_2^1 \\ \vdots & \vdots & \ddots & \vdots \\ I_1^b \times F_k^1 & I_2^b \times F_k^1 & \ldots & I_p^b \times F_k^1 \end{bmatrix} = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix}$$

The computation is done in time $t_2 = 4k^2$ clock cycles.

Step 3: the following $(k-1)$ shift operations are applied to the last $(k-1)$ vectors:

$$\begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix} => \begin{bmatrix} V_1 \\ V_2 << 1 \\ \vdots \\ V_k << (k-1) \end{bmatrix} = \begin{bmatrix} V_1' \\ V_2' \\ \vdots \\ V_k' \end{bmatrix} = \begin{bmatrix} v_{11}' & v_{12}' & \cdots & v_{1p}' \\ v_{21}' & v_{22}' & \cdots & v_{2p}' \\ \vdots & \vdots & \ddots & \vdots \\ v_{k1}' & v_{k2}' & \cdots & v_{kp}' \end{bmatrix}$$

The operation is performed in $t_3 = 5(k-1)$ clock cycles.

Step 4: the line of the featured plan is computed as follows:

$$C_b = \left[c_{b1} \; c_{b2} \; \ldots \; c_{b(p-k+1)} \right]$$

where, for $i = 1, 2, \ldots, (p-k+1)$:

$$c_{bi} = \sum_{j=1}^{k} v_{ji}'$$

The computation is done $t_4 = k + 1$ clock cycles.

Step 5: if $b < p - k + 1$ then $b \Leftarrow b + 1$ and go to ***Step 2***.

Step 6: The previous loop is repeated $p - k + 1$ times providing the result of applying the filter F^y on the “image” I with stride $s = 1$:

$$\mathscr{C}^f = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1(p-k+1)} \\ c_{21} & c_{22} & \cdots & c_{2(p-k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(p-k+1)1} & c_{(p-k+1)2} & \cdots & c_{(p-k+1)(p-k+1)} \end{bmatrix}$$

which is stored in the external memory. The execution of the transfer is in $t_6 = const \times (p-k+1)^2$.

Step 7: if $f < d$ then $d \Leftarrow d + 1, b \Leftarrow 1$ and go to ***Step 2***.

This loop is repeated d times resulting in the external memory a 3-dimension array of $(1 + (p - k)/s) \times (1 + (p - k)/s) \times d$ scalars.

The total execution time depends on the size of the input plan, the size of the filter and the number of features, and is:

$$t_{conv}(p, k, d) = t_{transfer} + t_{computation} =$$

$$t_{transfer} = (const \times (p^2 + dk^2 + d(p - k + 1)^2)) \in O(dp^2)$$

$$t_{computation} = d(4(p - 2)k^2 + (6p + 10)k - 4(p + k^3 + 4)) \in O(pk^2 d)$$

The convolution looks like an IO bounded function. In order to balance the data transfer with the computation we need to have a high bandwidth with MEMORY (the value of `const` must be small), and the overall DCNN computation must be organized with pooling layers applied before sending to MEMORY the feature plan. Thus, the weight of computation will be increased and the data to be transferred reduced.

5.2.2 Stride $s > 1$

For stride $s > 1$ the resulting 3-dimension array has the size:

$$(1 + (p - k)/s) \times (1 + (p - k)/s) \times d$$

because:

- vector C_i computed in ***Step 4*** is:

$$C_i(s) = [c_{b1} \underbrace{x \ \dots \ x}_{s-1} \ c_{b(s+1)} \underbrace{x \ \dots \ x}_{s-1} \ c_{b(2s+1)} \underbrace{x \ \dots \ x}_{s-1} \dots]$$

  where x stands for a meaningless value which must be removed from the final result
- in ***Step 5*** $b \Leftarrow b + s$ resulting in ***Step 6*** a $\mathscr{C}^f$ matrix with $1 + (p - k)/s$ lines only:

$$\mathscr{C}^f(s) = \begin{bmatrix} c_{11} & x \dots x & c_{1(s+1)} & x \dots \\ c_{(s+1)1} & x \dots x & c_{(s+1)(s+1)} & x \dots \\ c_{(2s+1)1} & x \dots x & c_{(2s+1)(s+1)} & x \dots \\ \vdots & \vdots \ddots \vdots & \vdots & \vdots \ddots \end{bmatrix}$$

The solution to provide a compact representation, by eliminating the xs, has two versions, one for the usual case when s is a power of 2 and another when s is of some value. In both cases we must add some sub-steps in ***Step 6***.

Version 1 for $s = 2^{integer}$: in ***Step 6*** we must use the scan operation `compact(i, j)` (see Sect. 4.3.5). If the stride is s then applying $log_2 s$ times the function `compact(i, j)` on each line of the matrix $\mathscr{C}^f(s)$ the meaningless values x will be eliminated.

The execution time for this step is:

$$t_{comp} = (1 + (p - k)/s) \times (2 + log_2 n) \times log_2 s$$

maintaining the execution time of ***Step 6*** dominated by the transfer.

Version 2 for s **of some value**: in ***Step 6*** we must add the following sub-steps:

Sub-step 6.1: The matrix $\mathscr{C}^f(s)$ is transposed. Results a matrix $T(\mathscr{C}^f(s))$ with lines full of xs and lines containing only c_{ij} scalars.

Sub-step 6.2: The matrix $T(\mathscr{C}^f(s))$ is compacted eliminating the lines of xs:

$$\mathscr{K}^f(s) = \begin{bmatrix} c_{11} & c_{(s+1)1} & c_{(2s+1)1} & \cdots \\ c_{1(s+1)} & c_{(s+1)(s+1)} & c_{(2s+1)(s+1)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Sub-step 6.3: The matrix $\mathscr{K}^f(s)$ is transposed resulting the final result for the filter f

$$\mathscr{S}^f(s) = \begin{bmatrix} c_{11} & c_{1(s+1)} & c_{1(2s+1)} & \cdots \\ c_{(s+1)1} & c_{(s+1)(s+1)} & c_{(s+1)(2s+1)} & \cdots \\ c_{(2s+1)1} & c_{(2s+1)(s+1)} & c_{(2s+1)(2s+1)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

which is a matrix s^2 times smaller than the matrix for $s = 1$.

The execution time for ***Step 6*** increases with

$$t_{6+} = t_{transpose} + 3p/s$$

Because, in our architecture $t_{transpose} \in O(p^2)$ the execution time of ***Step 6*** remains in the same magnitude order.

5.3 *Pooling Layer*

The input for pooling is a matrix of type I (see Eq. 6). The output is the matrix of

$$(1 + (p - k)/s) \times (1 + (p - k)/s)$$

```
/*********************************************************************
ALGORITHM NAME: Softmax
    - input: V = <x1, ..., xp>
    - output: V = <sigma_1(V), ..., sigma_p(V)>
*********************************************************************/
(1)     V <= V - redmax(V);
(2)     V <= V - lnLUT(redsum(expLUT(V)));
(3)     V <= expLUT(V);
```

Fig. 13 The softmax algorithm

components. The algorithm is similar to the one used for computing a feature plan in the convolutional layer for $s > 1$. Instead of a filter on the receptive fields are applied simpler functions. The differences are given by the lack of the filter, and occurs:

- in ***Step 2***, where instead of $I_i^b \times F_k^j$ is computed the sum $\sum_{i=b}^{b+k-1} x_{ji}$ or the maximum $MAX_{i=b}^{b+k-1} x_{ji}$
- in ***Step 4***, where is maintained $c_i = \sum_{j=1}^{k} v'_{ji}$ or is computed $c_i = MAX_{j=1}^{k} v'_{ji}$.

Thus, the computation time is evaluated in the same way, but refers to smaller input matrices.

5.4 Softmax Layer

For the softmax layer the exponential and logarithmic functions are computed using LUTs because the accuracy offered by this way is enough in the domain of NN. In our implementation we use a LUT for the logarithm, `logLUT`, stored in the data memory of CONTROLLER, and another LUT for exponentiation, `expLUT`, replicated in each of the n cells of ACCELERATOR.

According to the solution presented in Sect. 2.4 the algorithm on our accelerator is shown in Fig. 13. The execution time is $t_{softmax} = 9 + 4log_2n$. The loop $MAP \rightarrow REDUCE \rightarrow CONTROLLER \rightarrow MAP$ is closed two times, and both, the reduction and the distribution network are log-depth. In step (1) of the algorithm CONTROLLER sends back to the MAP array, through a log-stage pipe, the maximum value of the vector `V` received from a log-depth circuit from the MAP array. In the second step of the algorithm, `redsum` is received by CONTROLLER in log-time, and the logarithm is sent back in the same time to the array. Thus the acceleration of this layer is in $O(n/log\ n)$.

The latency introduced by the reduction operations can be avoided, as we did for matrix-vector multiplication, if the function is applied to a stream of vectors, $[V_1, V_2, \ldots, V_u]$, accumulated in the local memories distributed along the cells of the array. Then the values for $max_i = redMax(V_i)$ can be stored in the data memory of CONTROLLER. If $n \sim u$ then this computation is done in $O(u)$ time avoiding the contribution of $log_2 n$ for each max_i. Similarly, the values for $redsum(expLUT(V_i))$ are treated. Thus, the computation will be accelerated by $O(n)$.

6 Conclusions

1. The acceleration provided for each layer is in $O(n)$ for a n-cell accelerator. Sometimes, the constant associated to $O(n)$ is > 1, i.e., the acceleration is supra-linear, because the control, the transfer and the computation are done in parallel (see Sect. 5.1).

2. For MSRA, $\alpha = actualPerformance/peakPerformance$ in performing the computations associated to the stages of DNN is very high. Usually, $\alpha > 0.8$.

3. The data transfer between ACCELERATOR and MEMORY is transparent to the computational process. Thus, the effect of the bottleneck between ACCELERATOR and MEMORY is reduced.

4. The power consumption in our ***programmable system*** is 680 TGOPS/W[2] not far from the power consumption per Tensor operation provided by the TPU ***circuit***.

GFOPS/Watt is 2× higher than for many-cores, and 3× higher than for multi-cores.

5. But, we must pay attention to how the computational layers are interleaved with the data transfer stages in the implementation of an actual DNN. The main advantage in this respect for our architecture is the local memory in each cell of the MAP section. If this memory is big enough, then some data transfers can be avoided. Another advantage for our architecture is the possibility to transfer data in parallel with the computational process (see the algorithm for matrix-vector multiplication in Fig. 12). The overall α coefficient, taking into account also the transfers between the local memory in cells and MEMORY (see Fig. 8), decreases a little if the algorithms do ignore the possibility of transparent transfers.

6. Because MSRA has few characteristics similar to the Streaming SIMD Extensions (SSE) we must emphasize that the main differences consist of:

1. the control at the level of MSRA
2. the predicated execution according to the local state of each cell
3. the scan and reduction mechanism, which allows fast and efficient global vector to vector and vector to scalar operations
4. the large amount of local storage at the cell level, instead of the limited register file system in SSE

[2] This peak performance is achievable taking in consideration the operations performed simultaneously in the MAP section and in the REDUCE section (see Fig. 9). This happens, for example, when matrix-vector multiplication is performed.

5. data and programs in MSRA are provided through simple buffer-like memories, due to the high predictability of their content, unlike in the SSE system where data and programs are provided through the area and energy consuming cache memory system
6. search and scan instructions in the MSRA approach help advanced stream, list or sparse matrix operations.

References

1. Andonie, R., Malita, M.: The connex array as a neural network accelerator. In: Proceedings of the IASTED International Conference on Computational Intelligence, Banff, Alberta, Canada, 2–4 July, pp. 163–167 (2007)
2. Andonie, R., Malita, M., Stefan, M. G.: MapReduce: from elementary circuits to cloud. In: Kreinovich, V. (ed.) Uncertainty Modeling. Studies in Computational Intelligence, pp. 1–14. Springer, Cham (2017)
3. Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: efficient primitives for deep learning. https://arxiv.org/pdf/1410.0759.pdf (2014)
4. Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, pp. 1237–1242 (2011)
5. Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D.D., Henry, G., Pabst, H., Heinecke, A.: Anatomy of high-performance deep learning convolutions on SIMD architectures. SC. https://arxiv.org/pdf/1808.05567.pdf (2018)
6. Harris,, M.: Mixed-precision programming with CUDA 8. https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/ (2016)
7. Hennessy, J.L., Patterson, D.: Computer Architecture A Quantitative Approach, Sixth edn. Morgan Kaufmann (2019)
8. Jouppi, N.P., Young, C., Patil, N., Patterson, D., et al.: In-datacenter performance analysis of a tensor processing unit^{TM}. In: 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, 26 June. https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view (2017)
9. Kennedy, P.: Case study on the Google TPU and GDDR5 from Hot Chips 29. https://www.servethehome.com/case-study-google-tpu-gddr5-hot-chips-29/ (2017)
10. Kumar, A., Trivedi, M.: Intel scalable processor architecture deep drive. https://en.wikichip.org/w/images/0/0d/intel_xeon_scalable_processor_architecture_deep_dive.pdf (2017)
11. Li, C., Yang, Y., Feng, M., Chakradhar, S., Zhou, H.: Optimizing memory efficiency for deep convolutional neural networks on GPUs, SC '16. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, pp. 633–644. https://arxiv.org/ftp/arxiv/papers/1610/1610.03618.pdf (2016)
12. Lorentz, I., Malita, M., Andonie, R.: Evolutionary computation on the connex architecture. In: Proceedings of The 22nd Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2011), pp. 146–153 (2011)
13. Malita, M., Stefan, G.M., Thiébaut, D.: Not multi-, but many-core: designing integral parallel architectures for embedded computations. ACM SIGARCH Comput. Archit. News **35**(5), 32–38 (2007)
14. Malita, M., Stefan, G.M.: Map-scan node accelerator for big-data. In: 2017 IEEE International Conference on Big Data (BIGDATA), 4th Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery, 11–14 December 2017, Boston, MA, USA, pp. 3442–3447. http://users.dcae.pub.ro/~gstefan/2ndLevel/technicalTexts/S18203.pdf (2017)

15. Smith, R., Oh, N.: The NVIDIA Titan V preview—Titanomachy: war of the Titans. In: AnandTech. https://www.anandtech.com/show/12170/nvidia-titan-v-preview-titanomachy/2 (2017)
16. Stefan, G.M., Sheel, A., Mîţu B., Thomson, T., Tomescu, D.: The CA1024: a fully programmable system-on-chip for cost-effective HDTV media processing. In: Stanford University: Hot Chips: A Symposium on High Performance Chips, August 2006. https://youtu.be/HMLT4EpKBAw at 35:00 (2006)
17. Stefan, G.M., Malita, M.: Can one-chip parallel computing be liberated from ad hoc solutions? A computation model based approach and its implementation. In: 18th International Conference on Circuits, Systems, Communications and Computers, pp. 582–597 (2015)
18. Yuan, B.: Efficient hardware architecture of softmax layer in deep neural network. In: 2016 29th IEEE International System-on-Chip Conference (SOCC), pp. 323–326 (2016)
19. Convolutional Neural Network. 3 things you need to know. https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html (2019)
20. Cloud TPU. System Architecture. https://cloud.google.com/tpu/docs/system-architecture (2019)
21. The Future is Hybrid Trends in Computing Hardware. https://www.xcelerit.com/resources/the-future-is-hybrid-trends-in-computing-hardware/ (2017)

Progress in Neural Network Based Statistical Language Modeling

Anup Shrikant Kunte and Vahida Z. Attar

Abstract Statistical Language Modeling (LM) is one of the central steps in many Natural Language Processing (NLP) tasks including Automatic Speech recognition (ASR), Statistical Machine Translation (SMT), Sentence completion, Automatic Text Generation to name a few. Good Quality Language Model has been one of the key success factors for many commercial NLP applications. Since past three decades diverse research communities like psychology, neuroscience, data compression, machine translation, speech recognition, linguistics etc, have advanced research in the field of Language Modeling. First we understand the mathematical background of LM problem. Further we review various Neural Network based LM techniques in the order they were developed. We also review recent developments in Recurrent Neural Network (RNN) Based Language Models. Early LM research in ASR gave rise to commercially successful class of LMs called as N-gram LMs. These class of models were purely statistical based and lacked in utilising the linguistic information present in the text itself. With the advancement in the computing power, availability of humongous and rich sources of textual data Neural Network based LM paved their way into the arena. These techniques proved significant, since they mapped word tokens into a continuous space than treating them as discrete. As NNLM performance was proved to be comparable to existing state of the art N-gram LMs researchers also successfully used Deep Neural Network to LM. Researchers soon realised that the inherent sequential nature of textual input make LM problem a good Candidate for use of Recurrent Neural Network (RNN) architecture. Today RNN is the choice of Neural Architecture to solve LM by most practitioners. This chapter sheds light on variants of Neural Network Based LMs.

Keywords Statistical language modeling · Natural language processing · Artificial intelligence · Machine learning · Deep learning

A. S. Kunte (✉) · V. Z. Attar (✉)
College of Engineering Pune, Wellesley Road, Shivajinagar, Pune 411005, Maharashtra, India
e-mail: kas15.comp@coep.ac.in; kunte.anup@gmail.com

V. Z. Attar
e-mail: vahida.comp@coep.ac.in

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0_11

1 Introduction

Statistical Language modeling (LM), referrers to creation of an abstract model of the natural language content, that can be used for understanding the nuances of the language, automatically generating language contents or translating them to other language. During past three to four decades, LM has seen, a lot of attention by diverse research communities ranging from psychology, neuroscience, data compression, machine translation, speech recognition, linguistics to mention few. SLM has been applied to many NLP tasks including, but not limited to, automatic speech recognition, spelling correction, text generation, machine translation, syntactic and semantic processing, optical character recognition and handwriting recognition.

Statistical estimation techniques like N-gram models became popular choice, owing to the simplicity of construction, applicability across diverse set of languages and availability of ever increasing amount of on-line language resources which these knowledge impoverished but data optimal techniques strive on.

The success story of N-gram models kind of stymied research work on approaches based on linguistic knowledge. With the widely cited work of Bengio et al. [1] in 2003, Artificial Neural Networks (ANN) were first successfully applied to the LM, in order to address the data sparseness problem present in N-gram models. Besides predicting the next word Neural Network Language Model (NNLM) could also learn a real valued vector representation for every word in a predefined vocabulary. Right from their inception, NNLMs showed nearly equivalent results compared to N-gram counter part. These models also proved complimentary to the N-gram models when used as an ensemble model alongside some N-gram variant.

We discuss major milestones that shaped the advances in development of various Deep Neural Network based LMs. Initial focus of NNLM research was on devising better training procedures, reducing the training time and number of parameters required to train while keeping model performance unchanged. The increased computing power and availability of ever increasing amounts of linguistic resources meant, NNLMs soon became models of choice for many researchers. Researchers used Feed Forward Architectures and experimented with adding hidden layers to it in order to make it deep. These Deep NNLMs (DNNLM) were successful enough to create some degree of confidence in researchers for experimenting with the network architectures used for NNLMs.

Textual data used as input to LMs is sequential in nature. Recurrent Neural Network (RNN) architectures are found more appropriate to handle such sequential data, hence RNNLMs were soon developed. Today these LMs have proved to be the model of choice for LM, by many practitioners in the field. RNN based models were good at encoding long term dependencies but had known weakness that of vanishing or exploding gradient which meant training them using Back Propagating gradient was difficult. To address these issues researchers came up with modified RNN architectures like Long Short Term Memory (LSTM), Gated Recurrent Units (GRU). These architectures could address the long term dependencies found in the natural language content.

Traditionally LM was considered as a task of predicting sentence probability based on words as linguistic units which made these models restricted to fixed vocabulary. Recently researchers are experimenting with use of sub-word information to fuel the better training of LMs. These include use of Char Convolution Neural Network, QasiRNN to mention a few.

In the next section we understand the mathematical model of word level Statistical Language Modeling and also briefly discuss about the N-gram Language models which were very popular baselines for LM for long time. Then we discuss some important extensions to basic N-gram models. In the next section we look at NNLM, a model proposed by Bengio et al. [1], in an attempt to understand the use of Neural Networks Architecture to solve LMs. In next section we attempt to present various neural network architectures employed by researcher to solve LM. In the next section we discuss the standard evaluation metrics used by LM researchers and also list out the state of the art (STOA) shown by various Neural Network Language models proposed in the literature on popular benchmark data sets. In the concluding section we try to comprehend where the field of NNLM is moving currently.

2 Statistical Language Modeling

Language modeling is the art of determining the joint probability of occurrence of a sequence of linguistic tokens. We will consider word as linguistic token and hence will try to find out probability distribution of all sentences in a training corpus. This is equivalent of determining probability, $P(w_1, w_2, \ldots, w_n)$ of every sentence $w_1, w_2, \ldots, w_n$. Here w_i represents ith word in the sentence. Typically we determine the sentence probability with

$$P(w_1, w_2 \ldots, w_n) = \prod_{i=1}^{n} P(w_i|w_1 \ldots w_{i-1}) \tag{1}$$

So it becomes important to find out $P(w_i|w_1 \ldots w_{i-1})$. As i tend to take larger values, this computation becomes costlier. To simplify this computation, we typically consider Markov assumption, which for example is like saying, the probability of occurrence of ith word in sequence is independent of any previous context word but the last two i.e. w_{i-2}, w_{i-1}. This simplifies Eq. (1) to:

$$P(w_i|w_1 \ldots w_{i-1}) \approx P(w_i|w_{i-2}, w_{i-1}) \tag{2}$$

Equation (2) represents simple Language model referred to as tri-gram language model (3-gram). Despite of it's simplicity it proves effective for many practical applications. For the problems which require larger context histories, we can generalize it to N-gram model where preceding N − 1 length word sequence is considered in estimation of probability of current word:

$$P(w_i|w_1 \ldots w_{i-1}) \approx P(w_i|w_{i-n+1} \ldots w_{i-1}) \tag{3}$$

2.1 N-Gram Language Model

Equation (3), computes probability of occurrence of word (w) given the context window (h) of n − 1 previous words which is $P(w|h)$. This can be estimated using maximum likelihood estimate which simply uses word co-occurrence frequencies to estimate $P(w|h)$

$$P(w|h) \approx \frac{C(w,h)}{C(h)} \tag{4}$$

In Eq. (4), $C(w, h)$ denotes number of times sequence (h, w) is seen where as, $C(h)$ denotes number of times (h) is seen in the training dataset. Unigram (1-gram) being at lower bound for N-gram models where, $|h| = 0$ and consequent models being bi-gram, tri-gram and so on as we increase length of context window by one. The Length of context window plus one determines the order of model.

It was observed that many of the probability estimates for higher order N-grams come out to be zero. (simply because those word sequences do not occur in the training sets.) A technique called smoothing was successfully applied to overcome this issue. Smoothing works by redistributing probability estimates between seen and unseen (zero-frequency) events, by exploiting the fact that some estimates, mostly those based on single frequency, are greatly over-estimated. Chen et al. [2] give a detailed overview and empirical comparison of various smoothing techniques. As cited in [3], the modified Kneser–Ney smoothing (KN) is reported to provide consistently the best results among all the smoothing techniques, at least for word-based language models.

Simplicity in their overall make up, easy applicability to any domain as well as language are the significant advantages in favor of N-gram based models. Till date N-gram score over many advanced techniques due to above reasons. Advanced techniques mostly are more complicated and provide marginal improvements that are not critical over basic N-grams.

The Downside of the N-gram story was problem termed as data sparsity and their lack of ability to scale easily to longer context windows. The number of possible N-grams increase exponentially with length of context which makes them not suitable for capturing long term context patterns. The problem is more profound as training data increases, as much of the patterns from training data cannot be effectively represented by N-grams and cannot be discovered during training. Many advanced techniques were proposed In the next section we will look at some of the early language modeling techniques for completeness.

3 Extensions to N-Gram Language Model

Despite the indisputable success of basic N-gram models, it was always obvious that these models were not powerful enough to describe language at sufficient level. N-gram models solely depend on how much generalization is captured in the training set. As training size would increase exponentially to capture generalization, model parameters would also increase in similar order and the overall model quality will be adversely effected. Thus N-gram models lack ability to extrapolate from the information presented to it by training data.

As stated earlier, LMs have been studied by diverse research communities there exist different language modeling techniques and number of variations in each technique in the literature published. While it is out of scope of this work to describe all of these techniques in detail, we will at least make short introduction to the important techniques and provide references for further details.

Cache Based Many studies in speech recognition have empirically shown that there is significantly higher chance of repetition of the rare word if it has occurred in recent history. Cache models [4] are supposed to deal with this regularity, and are often represented as another N-gram model, which is estimated dynamically from the recent history of usually few hundreds of words and interpolated with the main (static) N-gram model. Cache models provide truly significant improvements in perplexity (sometimes even more than 20%), there exists a large number of more refined variants that can capture the same patterns as the basic cache models for example, topic models, latent semantic analysis based model [5], trigger models [6] or dynamically evaluated models [4, 7].

As models using variants of caching showed considerable improvement in PPL and relative simple construction, these were popular among LM community. Often in research papers these were compared to weak baselines like bi-gram or tri-gram models thus these improvements in PPL were prone to miss interpretation as pointed out by Goodman et al. [3] in his review of many variants of these models. In his paper he also explains how the errors may get locked into the cache model, as a wrongly recognized word may also creep into cache and further hurt word error rate despite a good deal of perplexity improvements.

A remedy to this as suggested by Goodman et al. [3] is to use user feedback to only store correct words in cache, but its not a practical solution. Advanced versions, like trigger models or LSA models were reported to provide interesting WER reductions, yet these models were not commonly used in practice.

It is thus important to be careful when conclusions are made about a LM technique just based on intrinsic metric.

Class Based Higher order N-gram face a unique problem of data sparsity as most of the higher order sequences are rarely show up in the training data. Introduction of equivalence classes may alleviate this problem in higher order N-grams. In the simplest case, each word is mapped to a single class, which usually represents several words. Next, N-gram model is trained on these classes. This allows better generalization to novel patterns which were not seen in the training data.

Improvements are usually achieved by combining class based model and the N-gram model. There exists a lot of variations of class based models, which often focus on the process of forming classes.

As suggested by Goodman et al. [3] class based models show moderate perplexity improvements but they give considerable improvements in word error rate especially when the training data is of smaller size. This makes class based models more attractive than cache based models.

The disadvantages of class based models include high computational complexity during inference (for statistical classes) or reliance on expert knowledge (for manually assigned classes). Also as reported in Goodman et al. [3] improvements tend to vanish with increased amount of the training data. They feel these models were less useful in production environments.

Structured The statistical language modeling was criticized heavily by the linguists from the very first day of its existence. Objections from the linguistic community usually address the inability of N-gram models to represent longer term patterns that clearly exist between words in a sentence. Words in a sentence are often related, even if they do not lie next to each other. N-grams can't effectively encode this due to the fact that they are nothing better than Finite State Machines which themselves are incapable of doing so. However, these patterns can be often effectively described while using for example context free grammars (CFG).

This was the motivation for the structured language models that attempt to bridge differences between the linguistic theories and the statistical models of the natural languages. The sentence is viewed as a tree structure generated by a context free grammar, where leaves are individual words and nodes are non-terminal symbols. The statistical approach is employed when constructing the tree: the derivations have assigned probabilities that are estimated from the training data, thus every new sentence can be assigned probability of being generated by the given grammar.

The advantage of these models is in their theoretical ability to represent patterns in a sentence across many words. Also, these models make language modeling much more attractive for the linguistic community. However, there are many practical disadvantages of the structured language models which may include computational complexity, sometimes unstable behavior, ambiguity due to possibility of many parse trees for single sentence, questionable performance when applied to spontaneous speech recognition. Also it requires large amount of manual work that has to be done by expert linguists.

Despite great research efforts, the results of these techniques remained questionable. However, it was certain that the question addressed by them of consideration of long context patterns in the natural languages was an important one for generating more intelligent models of natural language.

Maximum Entropy Maximum entropy (ME) model is an exponential model with a form

$$P(w|h) = \frac{e^{\sum_i \lambda_i f_i(w,h)}}{Z(h)} \tag{5}$$

where **w** is a word in a context **h** and **Z(h)** is used for normalizing the probability distribution represented by

$$Z(h) = \sum_{w_i \in V} e^{\sum_j \lambda_j f_j(w_i,h)} \tag{6}$$

Thus it can be viewed as a model that combines many feature functions $f_i(w, h)$. The problem of training ME model is to find weights λ_i of the features, and also to obtain a good set of these features, as these are not automatically learned from the data. Usual features are N-grams, skip-grams, etc.

ME models have shown big potential, as they can easily incorporate any features. Rosenfeld [8] used triggers and word features to obtain very large perplexity improvement, as well as significant word error rate reduction. Chen et al. [9] proposed model M, which is a class based maximum entropy model. This model is reported to have a state-of-the-art performance on a broadcast news speech recognition task [10], when applied to a very well tuned system that is trained on large amounts of data and uses state of the art discriminative loss trained acoustic models. The significant reductions in WER are reported against a good baseline language model, 4-gram with modified Kneser–Ney smoothing, across many domains and tasks. While unique algorithms for training ME models were developed by the speech recognition community, Mikolov in thesis [11] showed that ME models can be easily trained by stochastic gradient descent. In fact, ME models can be seen as a simple neural network without a hidden layer.

Thus, ME models can be seen as a very general theoretically well founded technique that has already proven its potential in many fields.

Neural Network Neural Network Language Model (NNLM) by Bengio et al. [1, 12] is set of two follow up papers which are considered as first notably successful attempt at using Neural Network based solution to LM problem. This is parametric approach to LM based on maximizing log likelihood of training data. The neural network architecture used had a single hidden layer with feed forward architecture. NNLM not only solves LM by finding probability distribution of word sequences but also learns to represent words as vector representation in a continuous space. This adds much value to NNLM as these vector inputs are useful in many tasks in NLP.

Now we will shift our focus to use if Neural Network Architecture to solve LM problem. To begin with in the next section we discuss how Bengio et al. [1], applied Neural Network Based model to solve work level SLM.

Table 1 Neural architectures applied to LM

Model used	NN architecture
Neural network language model	Single layer feed forward
Deep neural network language model	Deep feed forward
Vanilla RNN	Recurrent neural network
Class based RNN	Recurrent network classed output
RNN with LSTM cell	Long short term memory cell RNN
RNN with GRU cell	Gated recurrent unit RNN
Bidirectional RNN	Bi direction input RNN
Char aware neural LM	Convolutional network with LSTM RNN

4 Neural Network Based Language Modeling

Starting with Bengio et al. [1] in 2003 researchers have used various neural network architectures to build models to solve LM. Table 1 shows some of important variants of NN Architectures employed.

We will briefly review important neural network architectures applied by different researchers to LM problem.

4.1 Neural Network Language Model (NNLM)

Bengio et al. [1] introduced NNLM which was a parametric language model based on Feed Forward Neural Network Architecture. This model could simultaneously learn distributed representation of each word in terms of distributed word vectors and learns probability distribution for word sequences, expressed in terms of these representations. Figure 1 shows architecture of NNLM proposed in Bengio et al. [12] in a follow up paper.

As seen from the Fig. 1 the input context words are converted to 1-hot orthogonal vector representation which are projected to a lower dimensional space using shared vector P to reduce the input parameter size. If vocabulary size is of the order 200 K words, projection layer order is 100–300. This layer then provides input to hidden layer which applies non linearity in the form of hyperbolic tangent or sigmoid function. Finally output layer consisting of a softmax layer achieves the objective of Language model. Softmax layer computes:

$$p(w_t|w_{t-n+1}^{t-1}) = \frac{e^{h^T v'_{w_t}}}{\underset{w_i \epsilon V}{\Sigma} exp(h^T v'_{w_i})} \tag{7}$$

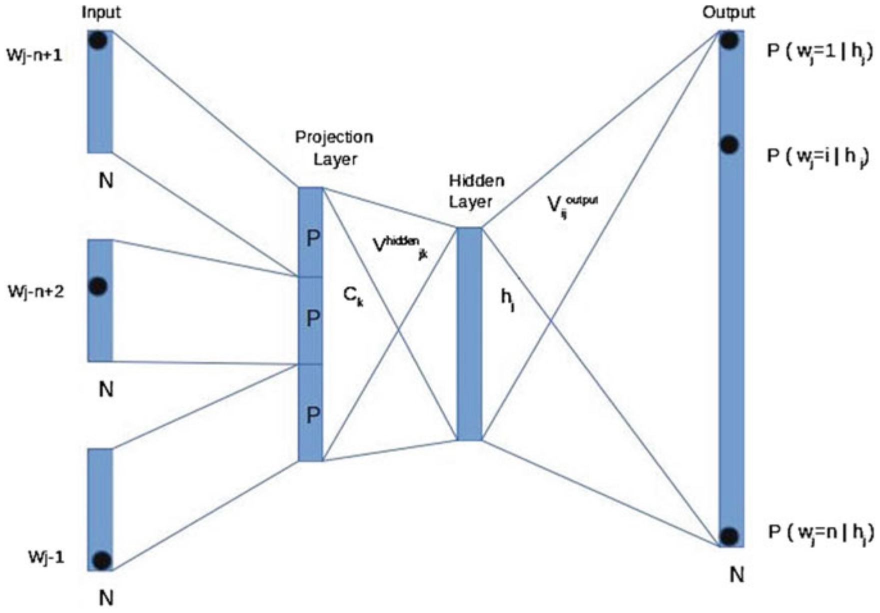


Fig. 1 Neural network language model

The inner product $e^{h^T . v'_{w_t}}$ computes the (unnormalized) log-probability of word w_t, which are normalized by the sum of the log probabilities of all words in V. The network thus satisfies the objective of Language Modeling. The network is trained using Error Back Propagation algorithm along with gradient descent for optimization.

In the follow up work Bengio et al. suggested strategies to improve the training process of the NNLM as they called the model using techniques like hierarchical output layer based on wordnet or pre trained clustering which reduced computations at output softmax layer considerably with a slight reduction in performance. Many of the follow up work in Neural LM seem to use a similar techniques suitably modified as required by the architectures they used and got results on similar orders. They also suggested use of idea of importance sampling, noise constrictive estimation as an option to full softmax calculations which would lead to considerable overall computational complexity. They also gave paralleled implementation of the training process which greatly reduced amount of time required by training process.

In 2012 paper, Arisoy et al. [13] experimented with NNLM architecture by adding more hidden layers to it which gave rise to Deep Neural Network LM. The additional hidden layers could improve the quality of LM trained without major increase in the training time as the dominating factor in NNLM's computational cost was the output softmax. They also experimented with around with number of hidden units and projection sizes.

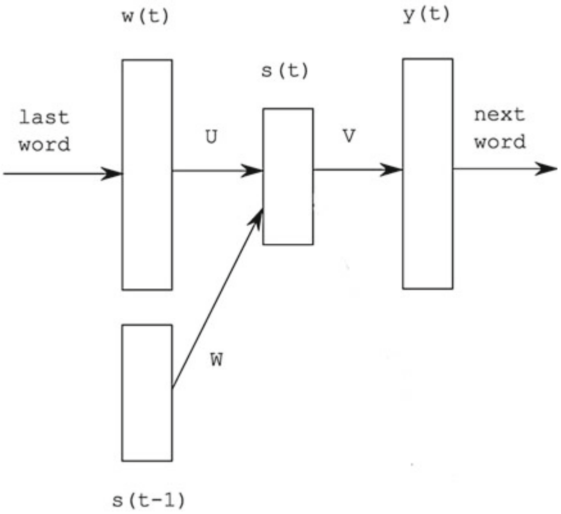


Fig. 2 Vanilla RNNLM architecture

Some researchers found these models were over fitting the training data, there were problems in deep networks like exploding and vanishing gradient BP algorithm. To counter aforementioned issues different regularization techniques were introduced for both shallow as well as deep feed forward neural network language models. The regularization's included L1, L2, Dropout, Blackout etc.

4.2 Recurrent Neural Network Language Models (RNNLM)

Recurrent Neural Networks (RNN) architecture have proven to be choice for sequence encoding applications and hence have become popular architectural choice for Deep Neural Network Language Models. Contrary to FNNs, RNN architecture, increases context length indefinitely using recurrent connection of hidden layers. Figure 2 shows Vanilla RNN architecture. The input layer consists of a vector **w(t)** that represents the current word w_t encoded as **1 of V** hot vector and vector **s(t − 1)** that represents output values in the hidden layer from the previous time step. After the network is trained, the output layer **y(t)** represents $P(w_{t+1}|w_t, s(t-1))$.

The network is trained by stochastic gradient descent using either usual error back propagation (BP) algorithm, or error back propagation through time (BPTT) [14].

The network is represented by input, hidden and output layers and corresponding weight matrices. Matrices U and W between the input and the hidden layer, and matrix V between the hidden and the output layer. Output values in the layers are computed as follows:

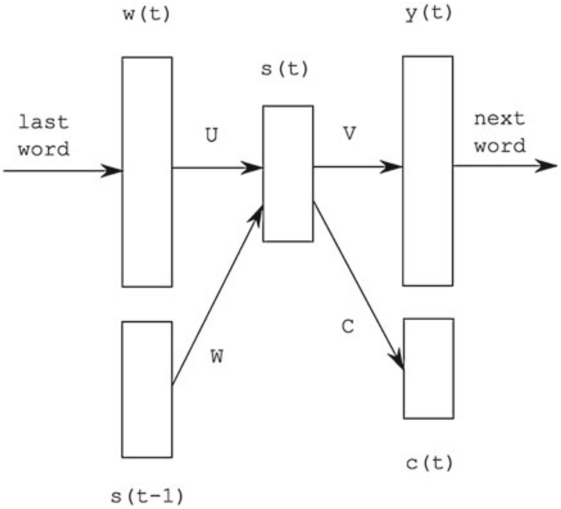


Fig. 3 RNNLM with classing

$$s_j(t) = f(\sum_i w_i(t)u_{ji} + \sum_l s_l(t-1)w_{jl}) \tag{8}$$

$$y_k(t) = g(\sum_j s_j(t)v_{kj}) \tag{9}$$

where $f(z)$ and $g(z)$ are sigmoid and soft-max activation functions respectively. This representation we do not include bias term for simplicity of representation. The output layer **y** represents a probability distribution of the next word $\mathbf{w_{t+1}}$ given the history. The time complexity of one training step will be proportional to:

$$\mathbf{O} = \mathbf{H} * \mathbf{H} + \mathbf{H} * \mathbf{V} = \mathbf{H} * (\mathbf{H} + \mathbf{V}) \tag{10}$$

where **H** is size of the hidden layer and **V** is size of the vocabulary. Figure 3 shows how to add class based output layer in RNNLM.

4.3 Long Short Term Memory Language Models (LSTMLM)

Recurrent Neural Network Architecture when used as RNNLM proved as good as NNLM and N-gram models with respect to the PPL improvements as shown by many researchers. These models were thought to capture the long term dependencies in linguistic data by nature of there recurrent behavior. But the researcher soon realised the Back Propagating error back words in time (BPTT) for training made the updates diminish as you go few steps back. Also since the model does not have control over what exactly to remember from the history context models, they could not show the potential improvements they promised. De Mulder et al. [15] surveyed about applications of RNN to Statistical Language Modeling. He mentions, three major drawbacks of basic RNNLMs training a RNN is known to be very slow, fixed number of hidden neurons and small context size in practice due to vanishing gradient problem experienced by BPTT algorithm.

The long short-term memory (LSTM) contains recurrently connected subnets, also called memory cells, were developed to increase the effective context size of RNN. In LSTM architecture extends basic RNN by replacing hidden unit with memory block. Each block contains one or more self-connected memory cells and three multiplicative units namely input, output and forget gates. These multiplicative units resembles memory cell operation of write, read and reset respectively. As long as the input gate has an activation value near zero, the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate. Figure 4 taken from [16] illustrates how a LSTM Cell is composed of and Fig. 5 the LSTMLM architecture.

The GRU (Gated Recurrent Unit) [17] is another variant of Recurrent Neural Networks and is pretty similar to an LSTM. GRU does not have a cell state and used the hidden state to transfer information. It have only two gates reset and update gate. The update gate acts similar to the forget and input gate of an LSTM. It decides what information to throw away and what new information to add. The reset gate is used to decide how much past information to forget. GRU has considerably less parameters and hence bit faster to train compared to LSTM architecture.

4.4 Bidirectional RNN

Basic RNN process data in forward direction that is history context is used to predict next token. For many NLP applications though, it is desirable to work the input in both the directions. This lead to development of the Bidirectional Recurrent Neural Network (BRNN) architecture.

Bidirectional RNNs (BRNNs) process the data in both directions with two separate hidden layers, which are then fed forward to the same output layer. As illustrated in the Fig. 5 from [16] a BRNN computes the forward hidden sequence $\overrightarrow{h}$, he backward hidden sequence $\overleftarrow{h}$ and the output sequence y by iterating the backward layer from t = T to 1, the forward layer from t = 1 to T and then updating the output layer. When we use LSTM for the hidden layer in BRNN it gives rise to Bi LSTM Network. Researchers have successfully used this architecture for LM.

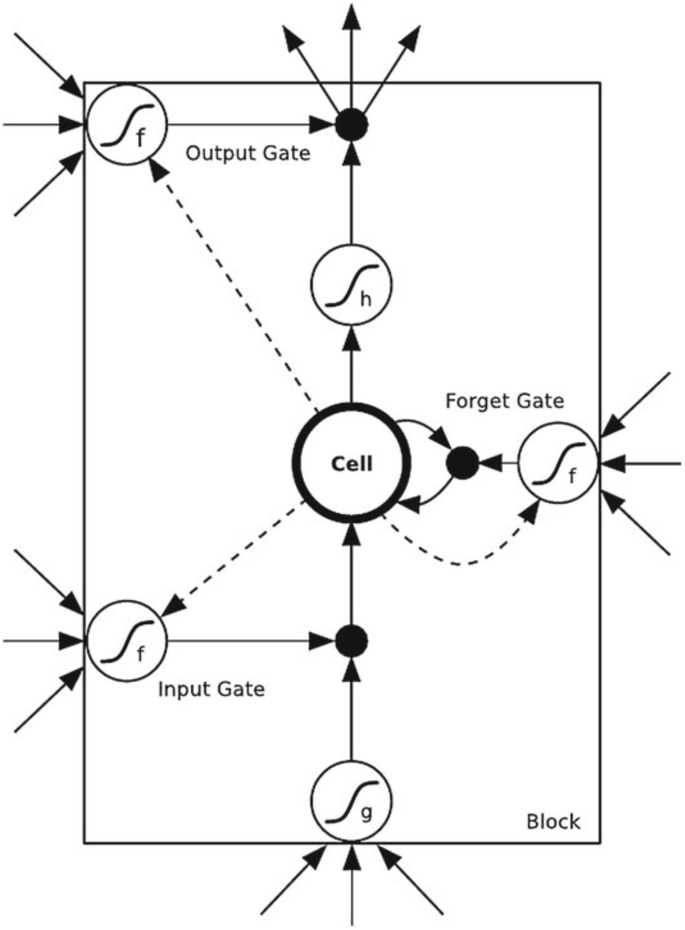
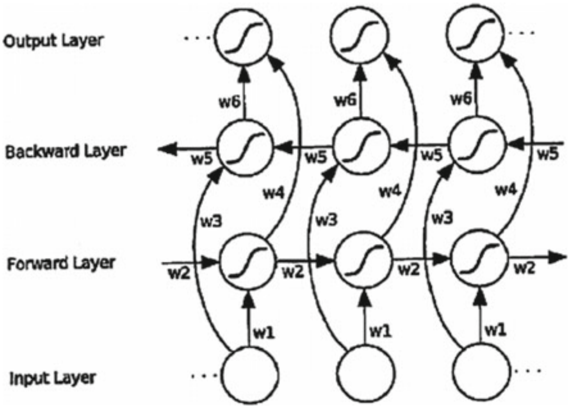


Fig. 4 Simple LSTM cell [16]

Fig. 5 LSTMLM architecture [16]



5 Milestones in NNLM Research

NNLM has seen much attention during past decade. In this section we present important research papers that have shaped advances in NLM research. We compare these research studies based on the neural network architecture, training algorithm, evaluation metric, special training recipes and corpus used.

Tables 2 and 3 show important experimental settings used in these studies. Based on these insights from the literature surveyed we feel any researcher who quest for a novel NLM technique must address following issues:

- Accelerating Training process for Large Scale LM based of Neural Networks pose unique challenges. This may involve using any one or combination of Biased Importance Sampling, Noise Contrastive Estimation, Hierarchical Soft-max, Self Normalizing Partition Function. One need to carefully characterize them.

Table 2 Various training methods used

Name of research publication	Neural network architecture	Training method used		
		Objective function	Optimizer stability	Training algorithm
Neural network model Bengio’ 03 [1]	FF	Log likelihood	Learning rate scheduling	BP
Adaptive importance sampling Bengio’ 06 [12]	FF	Log likelihood	Importance sampling	BP
Training large scale NNLM Mikolov ’11 [21]	RNN	Log likelihood	N.A.	BPTT
	RNNME	Discriminative	N.A.	
Deep neural network language model Arisoy’ 12 [13]	Deep FF	N.A.	N.A.	BP
Blackout: speeding up RNNLM with very large vocabularies Shiaho’ 16 [22]	RNN variant	Discriminative	Gradient clipping	BPTT
			Learning rate scheduling	
Noise contrastive estimation for large Vocabularies ’16 [23]	LSTM	Log likelihood	N.A.	BPTT
Limits of language Modeling ’16 [24]	LSTM	Discriminative	Gradient clipping	BPTT
	LSTM with Char cNN			

Table 3 Literature survey analysis

Paper	Evaluation metric	Training method		Corpus used	Task used
		Parameters	Speedup		
Neural network model bengio' 03 [1]	Intrinsic	$\sim V$	Parallel implementation	Brown corpus	N.A.
Adaptive importance sampling Bengio'06 [12]	Extrinsic	$\frac{\lvert V\rvert}{\log \lvert V\rvert}$	Biased importance	Switch board dataset	LVCSR
		$\log \lvert V\rvert$	Hierarchical decomposition		
Training large scale NNLM Mikolov'11 [21]	Intrinsic	$<< V$	Class based output layer	Penn tree bank	N.A.
Deep neural network language model Arisoy'12 [13]	Extrinsic	N.A.	N.A.	Wall street journal	ASR WER
Blackout: speeding up RNN language models with very large vocabularies Shiaho'16 [22]	Intrinsic	$<< V$	Blackout	PTB	N.A.
				1 Billion benchmark	
Noise contrastive estimation for large vocabularies [23]	Extrinsic	N.A.	N.A.	Wall street journal	ASR
Limits of language modeling [24]	Intrinsic	$Indp.of V$	Importance sampling	1 Billion benchmark	N.A.

- RNN LM and other variants of Deep Neural Networks based LM use BPTT during their training process, hence we must address problems like Vanishing Gradient, Exploding gradient. Reference [18] has shown how to deal with these problems.
- Literature shows that ensemble of various LMs always outperforms the base models. Right from the Non-parametric models like N-gram [3] and also NNLMs, RNN LM [1, 7, 12, 19, 20] have shown same characteristics. Its challenging to find a good ensemble of DNN which could be robust and easy for train.
- Another important issue with Neural Machine Translation model is that of handling rare word problem, Cho et al. [25] and Luong et al. [26].

- Another challenge is to study effectiveness of DNN LM for morphologically rich language and to come up with a combinational model of character level and word level granularity. One such study is [27] have proposed C2W (Compositional Character to word) model which can be used as preprocessing layer for LSTM based Deep LM.

6 Evaluation Metrics

Statistical language model can be evaluated with either an intrinsic metric or extrinsic metric. Intrinsic metric like perplexity reflect the quality of model learned compared to true model of language under consideration. Extrinsic metric like WER (Word Error Rate) on the other hand, reflects the applicability of the model to the task under consideration. Most of the researchers use either of them or both to analyze success of language model.

1. **The perplexity (PPL)** of word sequence w is defined as:

$$PPL = \sqrt[K]{\prod_{i=1}^{K} \frac{1}{P(w_i|w_1 \ldots w_{i-1})}} \tag{11}$$

   Perplexity is closely related to cross entropy between some held out test data and true model. It can be seen as exponential of average per-word entropy (geometric average) of test data. Smaller the value of PPL better is the model quality.

2. **The word error rate (WER)** is defined as:

$$WER = \frac{S + D + I}{N} \tag{12}$$

   where S is number of substitutions, D deletions and I insertions (each operation can change, delete or add a single word). The WER is defined for the lowest number of these operations that are needed to change the decoded utterance W_1 to the reference utterance W, which has N words.

3. **Bilingual Evaluation Understudy Score (BLEU)** is another extrinsic metric used in case of machine translation. It is a metric for evaluating a generated sentence to a reference sentence. A perfect match results in a score of 1.0, whereas a perfect mismatch results in a score of 0.0. The score was developed for evaluating the predictions made by automatic machine translation systems. It is not perfect, but does offer following compelling benefits:

   - It is quick and inexpensive to calculate.
   - Easy to understand.
   - Language Independent.
   - Correlate highly with human evaluation.

- It is widely adopted metric in case of machine translation, language generation, speech recognition, image captioning, text summarizing.

6.1 State of the Art PPL

There are a few benchmarks that people compare against for word-level language modeling. The difference in size, style, and pre-processing result in different challenges and thus different state-of-the-art perplexities.

- **Penn Treebank** (PTB) contains articles from the Wall Street Journal and the typical benchmark is the Mikolov-processed version. PTB has a vocabulary size of 10 K and training set of 890 K tokens. The challenge here is learning from a small amount of data. State-of-the-art is 47.7 ppl by Yang et al. [28]. They train with an AWD-LSTM (Merity et al. [29]) recurrent core containing 24 million parameters, incorporate dropout almost everywhere, and use dynamic evaluation [30] which allows the model to adapt to the test set. The state-of-the-art is 54.4 without this dynamic evaluation trick. A Kneser–Ney 5-gram achieves about 140 test ppl.
- **Billion Word Benchmark** is the biggest of all datasets used for benchmarking LMs. It is compiled from English-text news [31]. It contains 800 M words and a vocabulary size of 800K. Oleksii Kuchaiev [32] model is still state-of-the-art with 24.3 ppl. It is an LSTM with a bottleneck connection on the recurrent weights, containing 34 M parameters on the recurrent cell. In comparison, Kneser–Ney 5-grams obtain a perplexity of 67 [31]. The Billion Word Benchmark randomizes sentences which means models cannot take advantage of long term dependency or context. Furthermore random sentences are assigned to train and test. Since news articles can be very similar this does raise concerns about the fact that very similar sentences occur within train and test. As a result theres less of a strain to generalize, which means very large models which compress the training dataset do well on this task there is less of an emphasis to generalize.
- **WikiText-2** is a data set prepared by Salesforce (Merity et al. [29]) containing text from Wikipedia. The train and test constitute separate articles, which is nice. WikiText-2 is also small, it has a vocabulary of 30 K and contains 2M training tokens. Its kind of like a 2–3 times PTB and it does not appear to bring much additional research insight, as the main bottleneck is the small amount of data. The state-of-the-art is the same Yang et al. [28] paper, with 40.7 ppl.
- **WikiText-103** is another data set from Salesforce. This is much larger as the vocabulary size is 270 K and the training set contains 100 M tokens. The test set is the same as WikiText-2, but with the expanded vocabulary. The challenges here are: long-term temporal dependence and the large vocabulary. The state-of-the-art is 29.2 ppl from Rae et al. 2018 [33]. They propose a memory-inspired softmax layer, Hebbian Softmax, to better model rare words. The rest of the model consists of a very simple single-layer LSTM of 2048 units with input dropout and

memory-based adaptation at test time [34, 35]. The total model size is about 17 M parameters for the LSTM and 138M parameters for the input/output embedding matrix.

7 Conclusion

Statistical language modeling has been under research for more than 3 decades now. Simple language models like N-grams which have proved effective in practice and many More complex yet less practical models too. NLM has surely shown improvement over N-gram models. Since the field is matured to an extent that there are common recipes used by different researchers. With many variants of Deep NLMs available, still properly tuned RNNLM with LSTM or GRU cells beat many advanced NLMs with respect to PPL improvements. Recently researchers are concentrating on use of sub word information to fuel better Language models at word level. Using char level information makes these models independent of vocabulary.

References

1. Bengio, Y., Ducharme, R., Vincent, P., Jauvin, C.: A neural probabilistic language model. J. Mach. Learn. Res. **3**, 1137–1155 (2003)
2. Chen, J.G.S.: An empirical study of smoothing techniques for language modeling. In: Proceedings of the 34th Annual Meeting of the ACL (1996)
3. Joshua, T., Goodman, J.: A bit of progress in language modeling extended version, Machine Learning and Applied Statistics Group Microsoft Research. Technical Report, MSR-TR-2001-72 (2001)
4. Jelinek, F., Merialdo, B., Roukos, S., Strauss, M.: A dynamic language model for speech recognition. HLT **91**, 293–295 (1991)
5. Bellegarda, J.R.: A multispan language modeling framework for large vocabulary speech recognition. IEEE Trans. Speech Audio Process. **6**(5), 456–467 (1998)
6. Lau, R., Rosenfeld, R., Roukos, S.: Trigger-based language models: a maximum entropy approach. In: IEEE International Conference in Acoustics, Speech, and Signal Processing, ICASSP-93, vol. 2, pp. 45–48 (1993)
7. Mikolov, T., Karafiát, M., Burget, L., Cernockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Interspeech, vol. 2, p. 3 (2010)
8. Rosenfeld R.: Adaptive statistical language modeling: a maximum entropy approach. Ph.D. thesis, Carnegie Mellon University (1994)
9. Chen, S.F.: Shrinking exponential language models, In: Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Association for Computational Linguistics pp. 468–476 (2009)
10. Chen, S.F., Mangu, L., Ramabhadran, B., Sarikaya, R., Sethy, A.: Scaling shrinkage-based language models. In: IEEE Workshop on Automatic Speech Recognition & Understanding, ASRU 2009, pp. 299–304 (2009)
11. Mikolov, T.: Statistical language models based on neural networks. Ph.D. thesis, BRNO University of Technology, Faculty of information Technology (2012)
12. Bengio, Y., Schwenk, H., Senécal, J.S., Morin, F., Gauvain, J.-L.: Neural probabilistic language models. In: Innovations in Machine Learning, pp. 137–186, Springer (2006)

13. Arisoy, E., Sainath, T.N., Kingsbury, B., Ramabhadran, B.: Deep neural network language models. In: Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT, pp. 20–28 (2012)
14. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533 (1986)
15. De Mulder, W., Bethard, S., Moens, M.-F.: A survey on the application of recurrent neural networks to statistical language modeling. Comput. Speech Lang. **30**(1), 61–98 (2015)
16. Graves, A.: Supervised sequence labelling. In: Supervised Sequence Labelling with Recurrent Neural Networks, pp. 5–13. Springer (2012)
17. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using RNN encoder-decoder for statistical machine translation (2014). arXiv preprint arXiv:1406.1078
18. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. ICML **3**(28), 1310–1318 (2013)
19. Bengio, Y., Senécal, J.S.: Adaptive importance sampling to accelerate training of a neural probabilistic language model. IEEE Trans. Neural Netw. **19**(4), 713–722 (2008)
20. Bengio, Y., Boulanger-Lewandowski, N., Pascanu, R.: Advances in optimizing recurrent networks. In: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, pp. 8624–8628 (2013)
21. Mikolov, T., Deoras, A., Povey, D., Burget, L., Černockỳ, J.: Strategies for training large scale neural network language models. In: IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU) 2011, pp. 196–201 (2011)
22. Ji, S., Vishwanathan, S., Satish, N., Anderson, M.J., Dubey, P.: Blackout Speeding up recurrent neural network language models with very large vocabularies (2015). arXiv preprint arXiv:1511.06909
23. Zoph, B., Vaswani, A., May, J., Knight, K.: Simple, fast noise-contrastive estimation for large rnn vocabularies. NAACL HLT, pp. 1217–1222 (2016)
24. Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., Wu, Y.: Exploring the limits of language modeling (2016). arXiv preprint arXiv:1602.02410
25. Cho, S.J.K., Memisevic, R., Bengio, Y.: On using very large target vocabulary for neural machine translation (2014). CoRR arXiv:1412.2007
26. Luong, M.T., Sutskever, I., Le, Q.V., Vinyals, O., Zaremba, W.: Addressing the rare word problem in neural machine translation (2014). arXiv preprint arXiv:1410.8206
27. Ling, W., Luís, T., Marujo, L., Astudillo, R.F., Amir, S., Dyer, C., Black, A.W., Trancoso, I.: Finding function in form: compositional character models for open vocabulary word representation (2015). arXiv preprint arXiv:1508.02096
28. Yang, Z., Dai, Z., Salakhutdinov, R., Cohen, W.W.: Breaking the softmax bottleneck: a high-rank RNN language model (2017). arXiv preprint arXiv:1711.03953
29. Merity, S., Keskar, N.S., Socher, R.: Regularizing and optimizing LSTM language models (2017). arXiv preprint arXiv:1708.02182
30. Krause, B., Kahembwe, E., Murray, I., Renals, S.: Dynamic evaluation of neural sequence models (2017). arXiv preprint arXiv:1709.07432
31. Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., Robinson, T.: One billion word benchmark for measuring progress in statistical language modeling (2013). arXiv preprint arXiv:1312.3005
32. Kuchaiev, O., Ginsburg, B.: Factorization tricks for lstm networks (2017). arXiv preprint arXiv:1703.10722
33. Rae, J.W., Dyer, C., Dayan, P., Lillicrap, T.P.: Fast parametric learning with activation memorization (2018). arXiv preprint arXiv:1803.10049
34. Grave, E., Joulin, A., Usunier, N.: Improving neural language models with a continuous cache (2016). arXiv preprint arXiv:1612.04426
35. Sprechmann, P., Jayakumar, S.M., Rae, J.W., Pritzel, A., Badia, A.P., Uria, B., Vinyals, O., Hassabis, D., Pascanu, R., Blundell, C.: Memory-based parameter adaptation (2018). arXiv preprint arXiv:1802.10542

Index

W. Pedrycz and S.-M. Chen (eds.), *Deep Learning: Concepts and Architectures*, Studies in Computational Intelligence 866,
https://doi.org/10.1007/978-3-030-31756-0