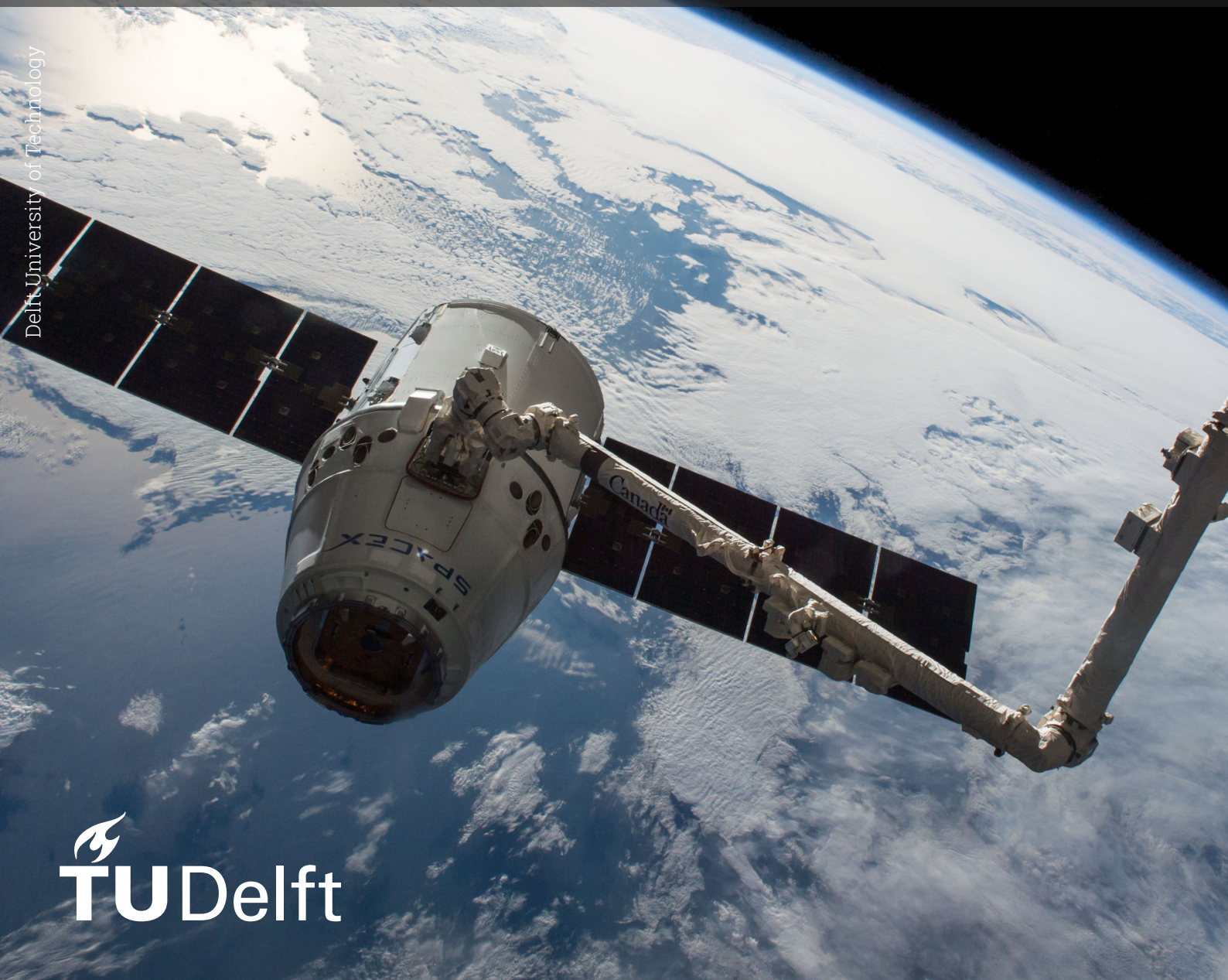
Sparse Temporal Convolutional Neural Networks for Keyword Spotting

Master Thesis

Peng Fu



Delft University of Technology

TU Delft

Sparse Temporal Convolutional Neural Networks for Keyword Spotting

by

Peng Fu

Student Name	Student Number
Peng Fu	5447860

Student number: 5447860
Project duration: Nov, 2022 - Dec, 2023
Thesis committee: Dr. Ir. Sijun Du — TU Delft, chairman
Dr. Ir. Chang Gao — TU Delft, supervisor

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Acknowledgments

After two years of study at TU Delft, I am about to graduate from the master's program.

I want to express my sincere gratitude to my thesis supervisor, Dr. Chang Gao, for his invaluable guidance and support throughout the research process. Dr. Gao's guidance was inspiring throughout the entire project. His patience in explaining complex concepts and his profound expertise in the field made each interaction both enlightening and constructive. His commitment to my academic and personal growth was evident in his countless hours providing valuable feedback and steering me in the right direction. Also, his professionalism and enthusiasm for research were truly inspiring. His insightful suggestions challenged me to think critically and creatively, pushing the boundaries of my understanding. The mentorship provided by Dr. Gao has been a cornerstone of my academic journey, and I am profoundly grateful for the wealth of knowledge and skills gained under his guidance.

Again, I am deeply grateful to Dr. Sijun Du for his pivotal role as the Chair of my thesis committee. His willingness to serve in this capacity and his continued support were critical to successfully completing my master's thesis.

A special note of appreciation goes to my best companion, Yuhang Jian, who has been my closest friend and an exceptional teammate throughout the past two years. Together, we navigated the complexities of numerous challenging course projects as a cohesive and dedicated team.

Finally, I would like to show my appreciation to my family. It's great to have their unlimited support for all those years of my study life.

Peng Fu
Delft, January 2024

Abstract

Keyword spotting (KWS) is an essential component of voice recognition services on smart devices. Its always-on characteristic requires high accuracy and real-time response. Also, low power consumption is another key demand for KWS devices. In previous research, neural networks have become popular for KWS tasks for their accuracy compared to traditional machine learning technologies. Among classical neural networks like recurrent neural networks (RNNs) and convolutional neural networks (CNNs), temporal convolutional networks (TCNs) have begun to catch attention recently. Moreover, studies related to sparsity are always an efficient method to deal with the growing model size issue for modern neural network designs. As a potential solution, in this work, a TCN model is trained for KWS on the Google Speech Command V2 dataset and achieves an accuracy of 94.1%. Based on that, two different sparsity are applied to the TCN model. One is temporal sparsity. By creating a Delta convolution layer, the Delta temporal convolutional network (DeltaTCN) achieves an accuracy of 93.6% with a 72% reduction in floating-point operations (FLOPS) compared to the original TCN model. Another is structural weight sparsity. By creating sparsity on the weight matrix of each convolution layer, the structural sparse temporal convolutional network (SSPTCN) achieves 93.6% accuracy with a 70% reduction in FLOPs and a 39% reduction in parameters.

Keywords- Temporal convolutional network (TCN); Keyword spotting (KWS); Temporal sparsity; Structural sparsity;

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
KWS	Keyword spotting
HMM	Hidden Markov Model
DNN	Deep neural network
TDNN	Time delay neural network
CNN	Convolutional neural network
RNN	Recurrent neural network
LSTM	Long short-term memory
GRU	Gated recurrent unit
TCN	Temporal convolutional network
SGD	Stochastic gradient descent
DeltaTCN	Delta temporal convolutional network
SSPTCN	Structural sparse temporal convolutional network
LFBE	Log-mel filter bank energies
MFCC	Mel-frequency cepstral coefficients
FLOPs	Floating point operations
MACs	multiply-and-accumulate operations

1

Introduction

Keyword spotting (KWS) [1] aims to detect pre-defined keywords in audio signals. Compared to common speech domains such as speech recognition [2, 3], speech synthesis [4], speech enhancement [5], and speaker recognition [6], KWS is relatively niche. But with the rise of intelligent assistants, smart speakers, and so on, KWS is receiving more and more attention from the industry. Its prevalent use revolves around facilitating hands-free control of mobile applications. With a primary emphasis on identifying wake-up words and recognizing common commands on mobile devices, achieving both immediate and accurate responses is crucial for Keyword Spotting (KWS). Nonetheless, the implementation of immediate and accurate KWS models remains a tough task, especially when faced with the real-time constraints of mobile devices equipped with restricted hardware resources.

Throughout the history of KWS, there have been diverse models for training KWS tasks. Traditional KWS tasks are realized by Hidden Markov Models(HMMs) [1]. With the development of deep learning, state-of-the-art KWS has switched to deep learning-based approaches with better performance [7]. Deep Neural Networks (DNNs) are commonly used together with compression techniques for KWS [8, 9]. The drawback of DNNs is that they often ignore the structure and context of the input [10]. Convolutional Neural Networks (CNNs) have become popular for KWS in the past few years [11]. Compared to DNNs, CNNs can achieve better performance and less model size [12, 13]. However, the drawback of CNNs is that they cannot model the context over the whole frame without wide filters or proper depth [10]. Recurrent Neural Networks (RNNs) are also a hot topic in KWS [14]. Similar to DNNs, RNNs also have the drawback on model input. RNNs perform modeling on the input features instead of learning the structure between continuous time and frequency steps. So, it is necessary to find a new model for KWS that can overcome the drawbacks of present deep learning models while maintaining the same performance.

Traditional neural networks are dense and suitable for current computing paradigms and GPU architectures [15]. Two issues have become increasingly vexing in recent years with deep learning models' increasing size and complexity. The first is memory and computational costs, which consume a lot of resources for training many new models [16]. The second is the generalization problem associated with the over-parameterization of large models, where the model tends to be overly sensitive to subtle input changes [17]. Although there are many methods to overcome some of the issues, such as L1/L2 weight regularization [18], dropout regularization variants [19] or data argumentation [20]. However, none of these approaches address the fundamental problem of over-parameterization: inefficient use of memory and computation.

This work has trained a Temporal Convolutional Network (TCN) for KWS, which avoids the drawbacks of CNNs and RNNs. Also, it exploited temporal sparsity and structural sparsity to reduce the use of memory and computation.

1.1. Motivation

For an extended period, Recurrent Neural Networks (RNNs) have been the predominant machine learning (ML) model employed in Keyword Spotting (KWS) systems for classification. Notably, RNN variants such as Gated Recurrent Unit (GRU) [21] and Long Short-Term Memory (LSTM) [22] networks have demonstrated high accuracy by utilizing a hidden state to retain information across different time steps. In more recent developments, Temporal Convolutional Networks (TCNs) have emerged as a successful alternative in KWS, achieving accuracy comparable to RNNs. For instance, in a study by Ibrahim et al.(2019)[23], TCN achieved an accuracy of 93.4% on the Google Brain dataset. Illustrated in Fig. 1.1, these networks leverage dilated convolutions with exponentially increasing dilation per layer, allowing them to cover a broader receptive field. The advantages of TCNs over traditional RNNs include enhanced parallelism, a flexible receptive field, and lower memory requirements for both training and inference.

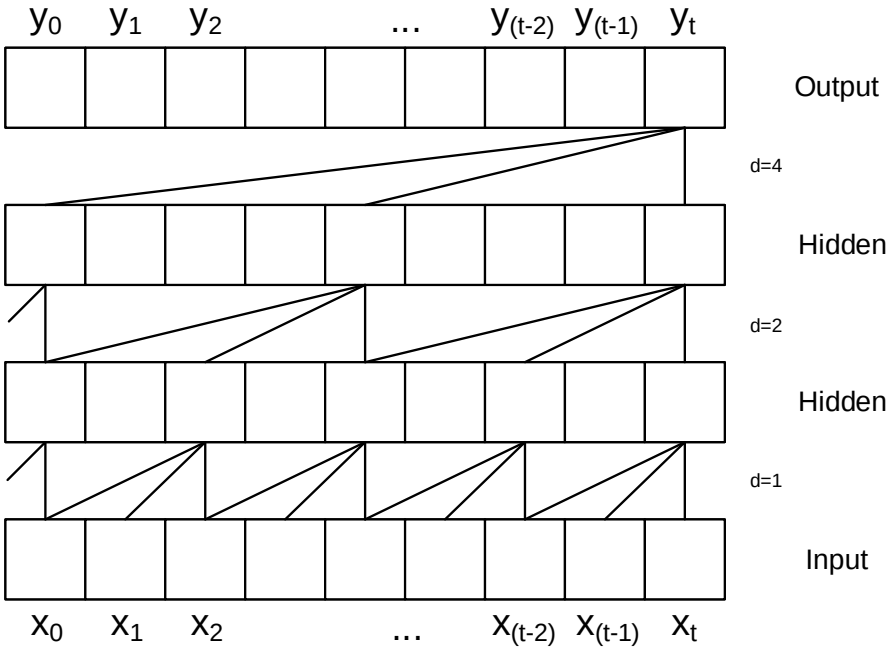


Figure 1.1: Inference for TCN networks

Historically, deep learning models have been characterized by their density and over-parameterization, sometimes leading to the memorization of random patterns in the data [24]. Studies have indicated that over-parameterized models are more suitable for training with stochastic gradient descent (SGD) compared to more compact representations [25]. However, this over-parameterization comes at the cost of increased memory usage and computational demands during both model training and inference. Sparse model representations offer a promising solution, particularly for inference on mobile devices, battery-powered devices, and in environments where cost considerations are paramount. Sparse models, with their reduced storage requirements, tend to streamline computation. Furthermore, over-parameterized models are prone to overfitting and may struggle to generalize to unfamiliar examples. Sparsification, therefore, serves as a form of regularization that can enhance model quality by effectively reducing noise within the model.

Recent research has concentrated on enhancing computational efficiency while upholding model accuracy, considering the computational demands of modern networks. State-of-the-art networks, such as Inception-V3 for object recognition [26], involved 5.7 billion operations and 27 million parameters for evaluation. Similarly, GPT-3 [27], a leading experimental natural language processing network, required a staggering 175 billion parameters for its evaluation. Moreover, the training of such deep neural models is progressively becoming more expensive, particularly for the largest language models that demand substantial data centers, incurring costs amounting to millions of dollars per training session. For instance, the training cost for GPT-3 mentioned earlier is approximately $4.6 million per session [28].

Therefore, exploring sparsity during training becomes imperative as a means to regulate and manage training costs.

1.2. Problem Description

Nowadays, there is a lot of research on keyword spotting, among which CNNs and RNNs are the hottest topics. Both of these neural networks have achieved excellent performance. However, there is relatively less research on TCNs [29]; however, it has gradually become a hot topic in recent years. Meanwhile, research on sparsity is also one of the focuses of deep learning [30, 31], but most of the research focuses on weight sparsity [32, 33] and activation sparsity [34, 32], and very few of them utilize temporal sparsity [35].

Until now, the discussion around sequence modeling in the realm of deep learning has predominantly centered on RNN architectures, including LSTMs and GRUs. S. Bai et al. [36] challenged this conventional perspective, asserting that it is outdated. They advocate considering Convolutional Networks as primary contenders for sequence data modeling. The study demonstrated that convolutional networks can outperform RNNs in various tasks while sidestepping common issues associated with recursive models, such as the challenges of gradient explosion/vanishing or inadequate memory retention. Additionally, the utilization of convolutional networks, as opposed to recurrent networks, enhances performance by enabling parallel output computation. Their proposed architecture is named TCN. TCNs can take a series of arbitrary lengths and output them as the same length. Causal convolution is used in the case of one-dimensional convolutional neural network (1D-CNN) architecture. A key feature is that the output at time t is only convolved with the elements that occurred before time t.

Temporally sparse sequential data is a type of data that rarely changes over time. Exploiting temporal sparsity to skip unnecessary arithmetic operations and memory access related to negligible input changes between adjacent time steps can help save energy consumption.

However, temporal sparsity is a random sparsification or unstructured sparsification. Structured weight sparsification clusters random sparsification, which can significantly reduce inference time on supported hardware, albeit with reduced accuracy compared to random sparsification.

This thesis combines the above two concepts and designs a sparse TCN model for KWS. Also, the thesis will explore how the threshold will affect the model sparsity and its consequences.

1.3. Objective

The main objectives of this work are,

1. To train a TCN model for keyword spotting.
2. To achieve sufficiently high temporal sparsity without suffering too much accuracy loss.
3. To study whether structural sparsification methods can facilitate the induction of sparsity.
4. To investigate the consequences of the method.

1.4. Contributions

The main contributions of this thesis are as follows:

- A standard TCN model for KWS was built. By scanning different combinations of parameters, the optimal result was selected as the baseline value for this project. The TCN model has an accuracy of 94.1% under the Google Speech Command V2 database with 116k training parameters.
- Exploiting temporal sparsity on top of TCN, the original one-dimensional convolutional layer is replaced using a sparse 1D convolution. The TCN model using temporal sparsity is called DeltaTCN. The DeltaTCN model achieves a mere 1.4% loss in accuracy but obtains a 72% reduction in FLOPs compared to a TCN model with the same specification parameters. Meanwhile, sparsity can be controlled by a user-defined threshold.
- Structured sparse is added to the TCN model, which is called Structural Sparse TCN (SSPTCN).SSPTCN achieves a 70% reduction in FLOPs and a 39% reduction in training parameters with only a 1.4%

loss in accuracy compared to TCN with the same parameter settings. Compared to DeltaTCN, an 80% reduction in training parameters and 44% reduction in FLOPs are achieved with the same accuracy.

1.5. Outline

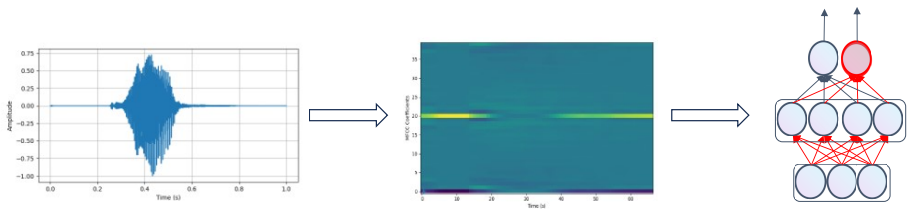
Chapter 2 reviews papers on the development of sequence modeling and introduces the Markov chain, RNNs, TDNNs, CNNs, and TCNs. Chapter 3 explains how DeltaTCN and structural sparsity are formed and its calculation. Chapter 4 is about the results and their discussion. Chapter 5 concludes the work and lists contents for possible future work.

2

Background

2.1. Keyword Spotting (KWS) Systems

The KWS system mainly detects pre-defined keywords. As shown in Fig. 2.1, a typical KWS system consists of two major parts: a feature extractor and a neural network classifier.



(i) Input speech signal (ii) Feature extraction (iii) Neural network

Figure 2.1: Framework of KWS system, from left to right: (i) Input speech signal (ii) Feature extraction (iii) Neural network

Feature extraction is the process of converting raw data into processable numerical features while preserving the information in the original dataset. In many cases, it will perform better for neural networks to train rather than use raw data directly [37]. Log-mel filter bank energies (LFBE) [38] and Mel-frequency cepstral coefficients (MFCC) [39] are the most commonly used methods to extract features. After LFBE or MFCC, the raw signal data will be translated from the time domain into a set of frequency-domain spectral coefficients. This enables the dimensionality compression for the input signal.

After extraction, the features are fed into a neural network-based classifier module, which generates the probabilities for the output classes. Traditional technologies for KWS mainly use HMMs [40]. With developing DNNs [7], researchers paid more attention to different neural networks. For state-of-the-art KWS systems, RNNs [41] and CNNs [42] are widely used as classifiers. The next section will discuss details about those KWS classifiers' technology.

2.2. Sequence Modeling

In traditional machine learning, data points are usually assumed to be independent and identically dispersed. But in many cases, such as linguistic, speech, and time-series data, each piece depends on the data before and after. This type of data is referred to as sequence data. A time series is a common example, where each point reflects an observation at some point in time, such as stock prices or sensor data. Sequences, such as audio waveform and weather data, are all examples of sequence data. In other words, video, audio, and images can all be considered sequential data to some extent. Here are a few basic examples of sequential data.

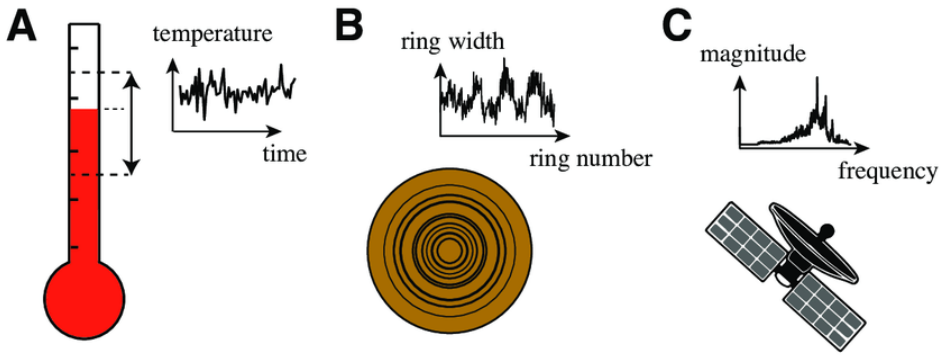


Figure 2.2: Examples of different sequential data types

In machine learning, learning for sequential data follows a similar concept of ordering.

A machine learning model that inputs or outputs a data sequence is called a sequence model. Text streams, audio clips, video clips, and time series data are all examples of sequence data. The analysis of sequence data such as text sentences, time series, and other discrete sequence data has facilitated the development of sequence models. The key factor in sequence modeling is that the processed data are no longer independent and identically distributed (i.i.d.) samples and are dependent on each other due to their sequence order. Sequential models are particularly popular in speech recognition, voice recognition, time series prediction, and natural language processing.

Sequential modeling is the process of generating a series of values from a set of input values. These input values may be time-series data showing how a variable has changed over time. The result produced may be a forecast of future demand. Another example is text prediction, where a sequence modeling algorithm predicts the next word based on a sequence of previous phrases and a set of pre-loaded conditions and rules.

2.3. Markov Chain

Prior to the emergence of convolutional networks, Markov models were widely employed for sequence modeling. In machine learning algorithms, a Markov chain represents a stochastic process in a state space, transitioning from one state to another. Notably, this process exhibits a ”memoryless” characteristic, where the probability distribution of the next state is solely dependent on the current state. The events that precede it in the time series are deemed irrelevant, defining the Markovian property. Markov chains find extensive applications as statistical models for various real processes.

A Markov chain, within the realm of mathematics, is a discrete-event stochastic process endowed with Markovian properties. In each step of a Markov chain, the system has the option to transition from one state to another or persist in its current state, or upon a probability distribution. This shift in state is referred to as a transfer, and the associated probability for each potential state change is termed the transfer probability. An illustrative example of a Markov chain is a random walk, where the state at each step corresponds to a point in a graph. During each step, the walker can move to any neighboring point, and the probability of moving to each point remains uniform, irrespective of the preceding path taken in the walk.

To summarize Markov chains in one sentence, the probability of a state transfer at a given moment depends only on its previous state. This may be a bit uncritical, but it greatly simplifies the complexity of the model, and thus, Markov chains are widely used in many time series models, such as RNNs and HMMs. Suppose the state sequence is

$$...x_{t-2}, x_{t-1}, x_t, x_{t+1}, x_{t+2}... \tag{2.1}$$

From the definition of the Markov chain, the moment x_{t+1} is only related to x_t.

$$P\left(x_{t+1} \mid ..., x_{t-2}, x_{t-1}, x_t\right) = P\left(x_{t+1} \mid x_t\right) \tag{2.2}$$

Since the probability of a state transfer at a given moment depends only on the previous state, the model of this Markov chain is fixed by requiring only the transfer probability between any two states in the system. Often, researchers use transfer matrices to calculate transfer probabilities. Each state in the state space appears in a column or row in the transfer matrix. Each cell in the matrix shows the probability of transitioning from a row state to a column state. In a state transfer matrix, the rows and columns are all possible states, and the corresponding position is the probability of transferring to the column state, given the known row state.

HMMs served as a key model in the early development of KWS. HMMs act as a probabilistic model capturing the temporal order, describing the generation process of random sequences of unobservable states through a hidden Markov chain. Simultaneously, they generate random sequences of observations by producing an observation associated with each state. The series of states randomly generated by the hidden Markov chain is termed a sequence of states. Each state contributes to the generation of an observation, forming a random sequence of observations. Each position within this sequence can be viewed as a specific moment in the overall process. Its form is defined below:

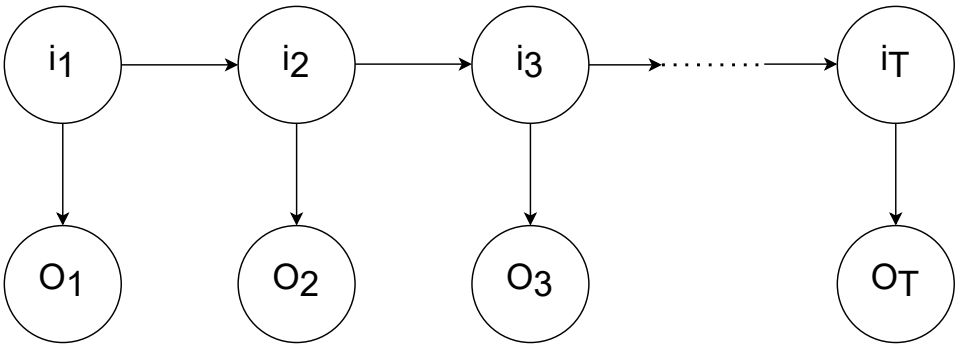


Figure 2.3: Example of a HMM

Much work has been done on KWS using HMMs in the last century. J. Robin Rohlicek et al.[43] proposed a KWS system using Gaussian HMMs. The paper suggested that the choice of features used, the covariance structure of the Gaussian model, and the transformations based on energy and confidence distributions can greatly impact performance. Richard C. Rose et al.[44] proposed a speaker-independent HMM keyword recognizer. The model was based on a continuous speech recognition model. It can reach an 82% probability of detection at a false alarm rate of 12 false alarms per keyword per hour. J. G. Wilpon et al.[45] proposed a KWS algorithm based on HMMs and achieved an accuracy of 99% on the Spanish database.

However, HMMs have their limits. They are hard to train and are computationally expensive during inference [46]. Also, the accuracy for HMMs in KWS is not comparable to state-of-the-art models, especially DNNs.

2.4. Recurrent Neural Networks

CNNs are said to be good in areas where the data are mostly static, and the input data are independent of each other, such as two-dimensional data like images, while for applications such as semantic recognition and time-series analysis, CNNs are slightly inadequate. This is because in applications such as semantic recognition, time series, etc., the data are sequential, and the front and back inputs are interrelated, so it is difficult for CNNs to consider the correlation between data in these applications, resulting in poor performance. For these sequential data, RNNs can be very good in addition to them. Based on this special structure, the RNNs have the ability of short-term memory, which preserves the correlation between data through ”memory”, so it is especially suitable for processing time-series data such as language, text, video, etc. This makes it particularly suitable for processing time series-related data, such as language, text, and video. Fig. 2.4 compares RNNs and feedforward Neural Networks(FNNs).

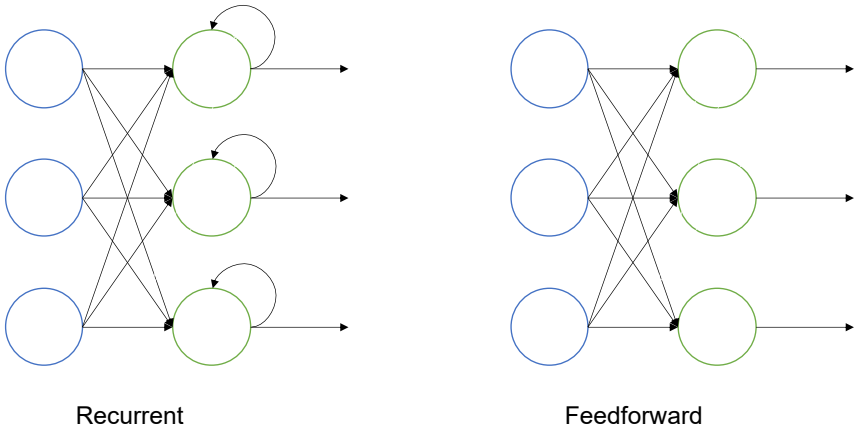


Figure 2.4: Difference between RNNs and FNNs in network structural

The structure of a traditional neural network is relatively simple: input layer - hidden layer - output layer. This is shown in the Fig. 2.5 below:

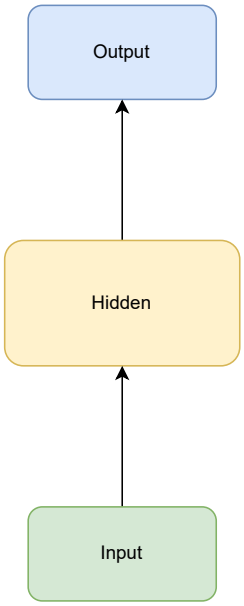


Figure 2.5: Example of traditional neural network structure

The biggest difference between an RNN and a traditional neural network is that each time, the output of the previous one is brought to the next hidden layer and trained together. This is shown in the Fig. 2.6 below:

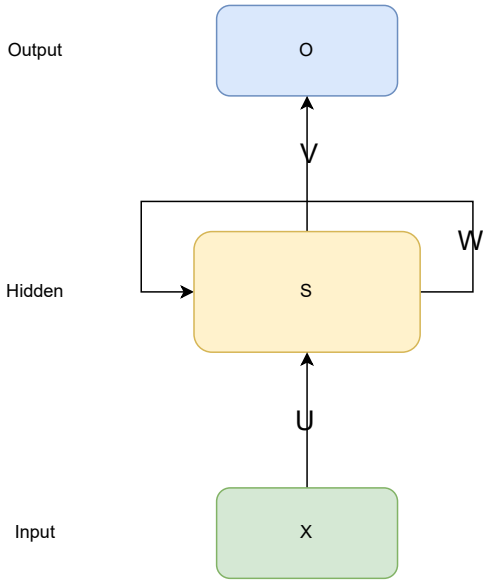


Figure 2.6: Example of a RNN structure

Let X be a vector denoting the values of the input layer, and S represent the values of the hidden layer in an RNN. The weight matrix U connects the input layer to the hidden layer, while O is a vector capturing the values of the output layer, and V represents the weight matrix from the hidden layer to the output layer. In an RNN, the hidden layer's value S is influenced not only by the current input X at this time but also by the value of the preceding hidden layer S. The weight matrix W utilizes the value of the hidden layer from the previous time as the weight for the input at the current time.

Expanding the Fig. 2.6 gives the timeline expansion of the RNNs. The timeline expansion is shown as follows:

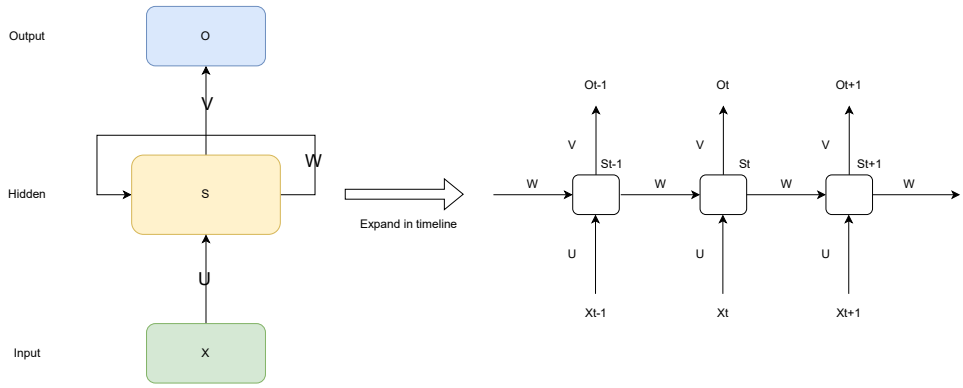


Figure 2.7: Expansion of RNNs in timeline

After the RNNs receive the input X_t at moment t, the value of the hidden layer is S_t, and the output value is O_t. The most critical point of the RNN network is that the value of S_t depends not only on X_t but also on S_{t-1}. Here, the calculation of the RNNs is represented by the following equation:

$$O_t = g(V \bullet S_t) \tag{2.3}$$

$$S_t = f\ (U \bullet X_t + W \bullet S_{t-1}) \tag{2.4}$$

Long short-term memory (LSTM) is one of the most famous types of RNNs. LSTM is a special RNN structure that is capable of learning long-term dependencies. Proposed by Hochreiter and Schmidhuber (1997) [47] and improved and generalized by many in the following work, LSTM performs very well

on a wide variety of problems and is now widely used. All RNNs have a chain repetition module of the neural network. In a standard RNN, this repetition module has a very simple structure, e.g., a single tanh layer. the LSTM has a similar chain structure, but the repetition module has a different structure. Instead of a single neural network layer, there are four, and they interact very specifically.

Gated recurrent Unit (GRU) [21] is also a kind of RNN. Like LSTM, it was proposed to solve the problems of long-term memory and gradient in backpropagation, etc. The actual performance of GRU and LSTM is similar in many cases, and the reason for choosing GRU is that it is simpler to compute. The use of GRU can achieve comparable results compared to LSTM and is easier to train compared to it, which can largely improve training efficiency so that GRU will be preferred in many cases.

Due to the advantages of RNNs on sequential data, RNNs have long been a popular model for KWS tasks.

2.5. Time Delay Neural Network

Time delay neural network(TDNN) has been proven efficient in dealing with long-range temporal dependencies. It is first proposed in [48]. TDNN was originally developed for phoneme recognition. The TDNN-based modeling framework is suitable for dealing with the inconsistency in the time length of the sequence of feature vectors possessed by speech. The output of each hidden layer is temporally extended, i.e., the input received by each hidden layer has the output of the previous layer at a different time. Speech is important in considering the long-time correlation of contexts. TDNNs have the advantage of observing longer time than traditional DNNs and are not slower than DNNs in training and decoding.

Two distinctive features of TDNN are dynamic adaptation to time-domain feature variations and fewer parameters. While traditional DNNs have input layers connected one by one with the hidden layer, TDNN makes a slight change here, i.e., the features of the hidden layer are not only related to inputs at the current moment but also to inputs at future moments.

Here is an example TDNN model from [48].

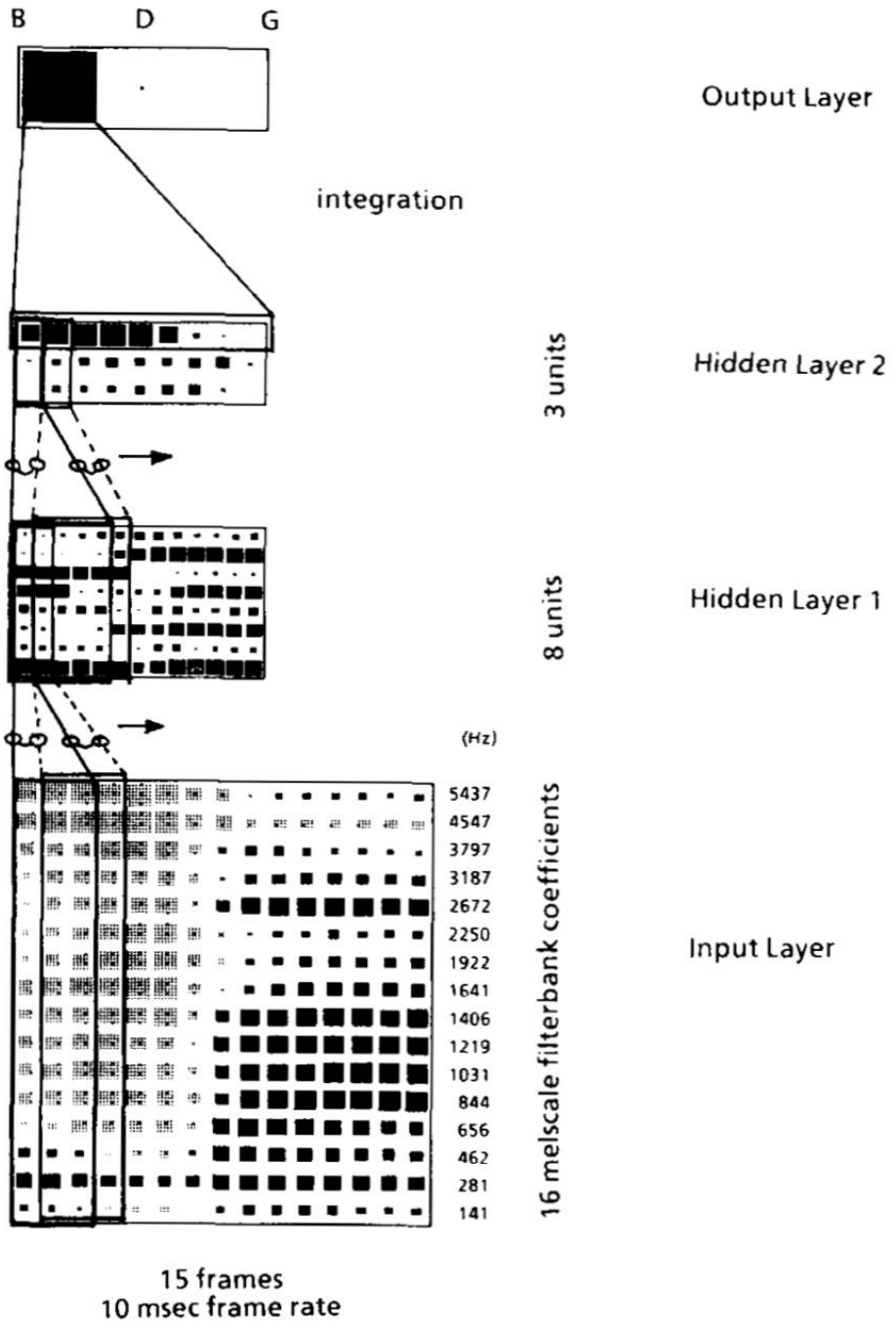


Figure 2.8: A 3 layer TDNN model, adapted from [48].

In [48], a three-layer TDNN is proposed for the task of speech recognition of letters ”B”, ”D”, and ”G,” as shown in the above Fig. 2.8. The input in the Fig. 2.8 is 15 frames of audio features; each frame feature is a Mel coefficient of length 16; here, the delay value is set to 2, i.e., the input of the above layer of the network consists of three consecutive moments of the below layer, it can be seen that the number of neurons in Hidden Layer 1 is 8. Since weight sharing is applied in TDNN, i.e., the weights of the same delayed position are shared, the input layer and Hidden Layer 1 have 48 weights between them. In Hidden Layer 2, a delay value of 4 was set, and the number of neurons in this layer was 3. Setting a larger delay value allows the network to see a larger range of time domain features. Finally, the output probability distribution of the three letters can be obtained by putting Hidden Layer 2 through an MLP.

Like other neural networks, time-delay neural networks operate with multiple interconnected layers

consisting of clusters [48]. These clusters are meant to represent neurons in the brain, and like the brain, each cluster only needs to pay attention to a small region of the input. The proto-typical TDNN has three layers of clusters: one set for inputs, one for outputs, and an intermediate layer that handles the operations of the inputs through filters. Due to their sequential nature, TDNNs are implemented as FNNs rather than RNNs.

To achieve time-shift invariance, a set of delays is added to the input (audio files, images, etc.) to represent the data at different points in time. These delays are arbitrary and application-specific, which usually means that the input data is customized for a specific delay pattern. Much work has been done to create adaptive delay TDNN. This manual tuning was eliminated. Delay attempts to add a time dimension to RNNs with sliding windows or networks not present in multilayer perceptrons. The combination of past inputs with existing inputs makes the TDNN approach unique. A key feature of TDNNs is the ability to express a relationship between inputs in time. The relationship can result from a feature detector and is used within the TDNN to recognize patterns between delayed inputs.[48]

TDNN has a wide range of applications as an early neural network structure.TDNN was initially used to solve problems in speech recognition, focusing on shift-invariant phoneme recognition[48]. TDNN is well suited to speech tasks because spoken words are rarely uniform in length and difficult to segment accurately. By scanning past and future voices, TDNN can model key elements of that voice in a time-shift invariant manner. This is especially true when the sound is masked by reverberation[49][50].

TDNN has been effectively applied to compact, high-performance handwriting recognition systems. Translation invariance is also applied to spatial patterns (x/y-axis) in offline handwriting recognition of images[51]. TDNNs were also successfully used in early demonstrations of audio-visual speech, where the speech sounds are complemented by visually reading lip movement[52].

The most popular TDNN application in recent years is ECAPA-TDNN[53]. ECAPA-TDNN was proposed by Desplanques et al. at the University of Gothic, Belgium, in 2020, and by introducing the SE (squeeze-excitation) module as well as the channel-attention mechanism, the scheme achieved first place in the International Voice Recognition Competition (VoxSRC2020).

Kalantari et al.[54] investigated how sparse input to time-delay neural networks (TDNNs) could significantly improve the learning time and accuracy of TDNNs for time series data. The study demonstrated that learning time and accuracy can be greatly improved by applying sparse input transformation layers to TDNN.

2.6. Convolutional Neural Networks

Traditional neural network layers are fully connected; if the number of sampled data layers is large and the input is high-dimensional, the number of parameters may be astronomical. Therefore, another way must be found to process data information such as pictures, videos, audio, and natural languages more effectively. After years of unremitting efforts, some effective methods and tools have been found [55, 56, 57]. Among them, CNNs are typical representatives. Fig. 2.11 shows a classic CNN example named AlexNet.

CNNs is a feed-forward neural network; the very first idea was proposed in the BP algorithm in 1986 [58]. In 1988, LeCun proposed the LeNet-5 model[59], which can be regarded as the prototype of CNNs. After that, research related to CNNs has been in the doldrums for two reasons: first, researchers realized that multilayer neural networks were extremely computationally intensive for BP training, which was utterly impossible with the hardware computing power at that time; second, shallow machine learning algorithms, including SVM, also started to emerge. In 2006, Hinton [60] brought back the attention to CNNs. 2012, CNN-based AlexNet won the ImageNet competition [61].

CNNs consist of one or more convolutional layers and a top fully connected layer (corresponding to classical neural networks) but also include association weights and pooling layers (Pooling Layer), etc. CNNs can give better image and speech recognition results than other deep learning architectures. This model can also be trained using a backpropagation algorithm. CNNs can achieve higher performance with fewer parameters than deep, feed-forward neural networks.

The model will have three major parts for a typical CNN network: convolutional layer, pooling layer, and

fully connected layer.

Convolutional layer The convolutional layer plays a crucial role in transforming the input image to extract relevant features. This transformation involves convolving the image with a kernel, as illustrated in Fig. 2.9. The kernel, often referred to as a convolution matrix or mask, is a smaller matrix with a height and width smaller than that of the input image. It slides over the height and width of the input image, and at each spatial location, the dot product of the kernel and the image is computed. The distance covered by the kernel in each slide is known as the step length. The resulting image is termed the convolution feature.

In convolution operations, the number of channels in the kernel needs to match that of the input image. The convolution layer is completed with the application of an activation function, introducing nonlinearity to the output. Common choices for activation functions in convolutional layers include the ReLU function or the Tanh function.

Fig. 2.9 depicts a simple convolutional layer where a 6x8 input image is convolved with a 3x3 kernel, assuming a stride of 1, to produce a convolution feature of size 6x8. An activation function is then applied to generate the output, often referred to as a feature map.

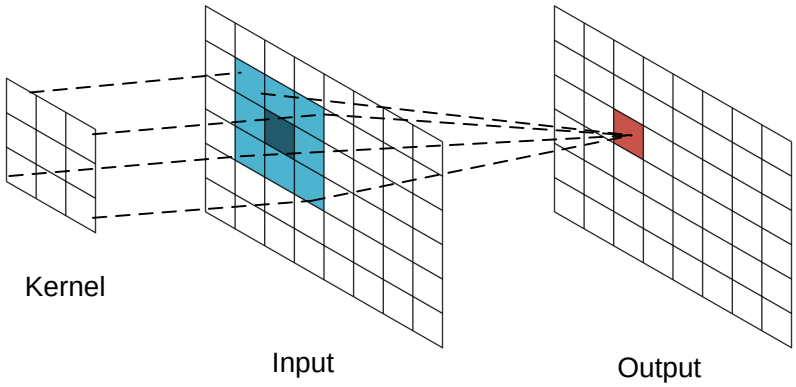


Figure 2.9: Example of a Convolutional Layer with a 3x3 kernel and a 6x8 input.

Pooling layer The pooling layer serves the purpose of reducing the size of the input image, typically following a convolutional layer in CNNs. Pooling layers contribute to speeding up computations and enhancing the robustness of certain detected features. Similar to convolutional operations, pooling operations involve the use of kernels and strides. In the provided example, a 4x4 input image with a stride of 2 is pooled using a 2x2 filter, resulting in a merged image. Maximum pooling and average pooling are CNNs' most commonly used pooling methods. The structures of two pooling layers are listed in Fig. 2.10. Maximum pooling: In maximum pooling, the maximum value is selected from each face slice of the element map to create a reduced map. Average pooling: The average value is selected from each face slice of the element map to create a reduced map.

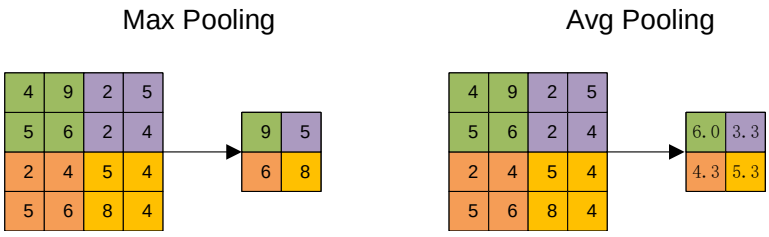


Figure 2.10: Two different types of pooling layer

Fully-connected (FC) layer Located at the end layer of CNNs, the fully connected layers play a pivotal role. The feature maps produced by preceding layers are flattened into a vector, which is then

input into the fully connected layer. This layer is important in capturing intricate relationships among higher-level elements. As a result, the output from this layer is a one-dimensional feature vector.

In Fig. 2.12, a straightforward CNN architecture is presented for binary image classification. This network is designed to categorize input into two distinct classes. It operates on RGB images with dimensions 32x32x3, and the output size is 2, corresponding to the number of classes being classified. The initial layer is a convolutional layer featuring 5x5x3 kernels. Subsequently, a second layer constitutes a maximum pooling layer with a 2x2 kernel size. Following this, the third layer comprises another convolutional layer with 5x5x3 kernels, and the fourth layer is a maximum pooling layer with a 2x2 kernel size. The output is then flattened into a vector and directed into the last two layers, both of which are fully connected.

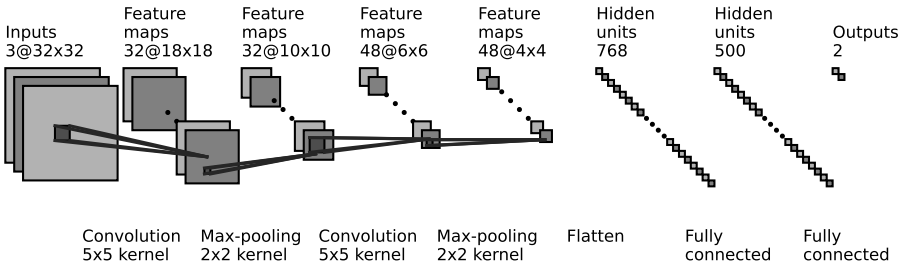


Figure 2.11: Network architecture of AlexNet [61]

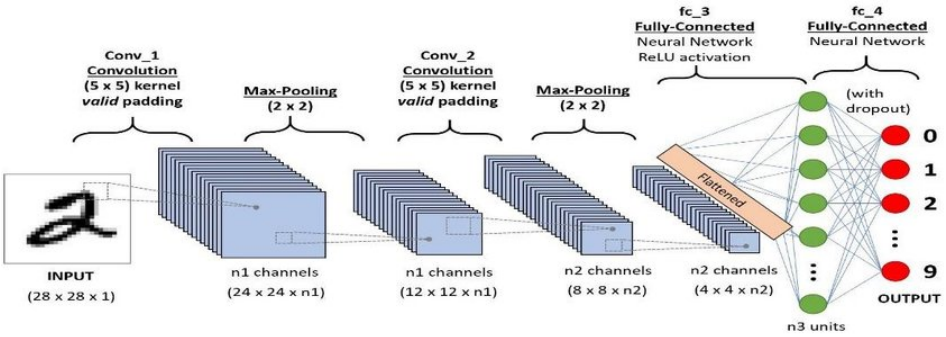


Figure 2.12: A CNN sequence to classify handwritten digits, adapted from [62]

2.7. Temporal Convolutional Networks

CNN has almost become the icon of deep neural learning and is mainly used in image processing. Also, one-dimensional CNNs can be used for mining temporal aspects. However, even though the theoretical analysis shows CNN can handle temporal prediction, it is not very effective in real practice. In this case, the TCNs were introduced to solve the temporal problems.

TCN has two principles: 1) it employs causal convolutions, ensuring that there is no information ”leakage” from the future to the past; 2) the architecture is capable of processing sequences of variable lengths and mapping them to output sequences of equivalent lengths.

1D Convolutional Network In a one-dimensional convolutional network, a 3-dimensional tensor is accepted as input, and the output is a 3-dimensional tensor. The TCN implementation specifically involves an input tensor shaped as (batch_size, input_length, input_size) and produces an output tensor shaped as (batch_size, input_length, output_size). Notably, as each layer in TCN maintains

consistent input and output lengths, the disparity lies only in the third dimension of the input and output tensor.

Causal Convolutions A TCN network follows two fundamental principles: first, it ensures that the network produces an output of the same length as the input, and second, it avoids any information leakage from the future into the past. To meet the requirement of producing an output of the same length as the input, TCN employs a 1D convolutional network architecture. Each hidden layer is designed with the same length as the input layer, and zero padding of a certain length is added to maintain consistency across subsequent layers. To address the second rule, TCN utilizes causal convolutions, ensuring that the convolutional operation at time t involves only elements from time t and earlier in the preceding layer. In fact, a TCN network can be considered as the combination of a 1D fully connected network (FCN) and causal convolutions.

Dilated Convolutions Dilated convolution is a technique to expand the kernel by inserting zeros between its consecutive elements. A dilated convolution will enable the network to have a larger receptive field without increasing the number of parameters. It is possible to view all the inputs as they reach the bottom along the blue line from the top to the bottom, which means that the output prediction (at time T, for example) uses all the inputs in the valid history data.

In an ideal predictive model, the output entries depend on the previous input entries, i.e., all entries with indexes less than or equal to itself. Full history coverage is achieved when the size of the receptive field is equal to the input size. When there is only one layer of indexes, the size of the receptive field is equal to the kernel size. If the receptive field needs to be expanded, then multiple stacking layers are required. This is shown in the Fig. 2.13.

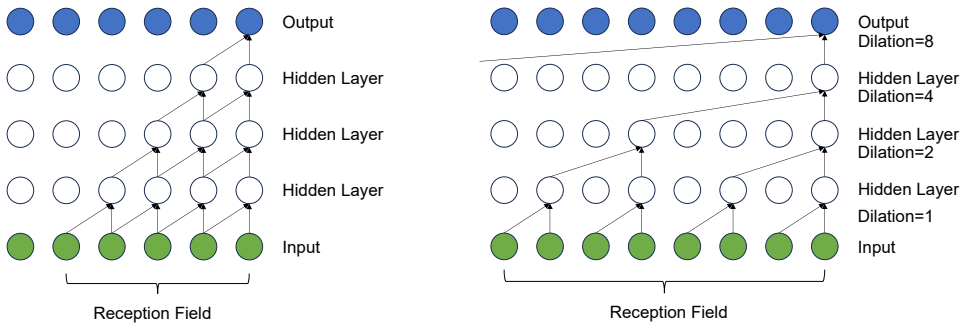


Figure 2.13: A dilated layer to expand receptive field, adapted from [36]

Generally, for a 1D convolutional network with n layers and a kernel size of k, the receptive field r can be calculated as follows:

$$r = 1 + n \times \ (k - 1) \tag{2.5}$$

In reverse, to achieve full coverage, the number of layers can be solved by input length and kernel size as follows:

$$n = \lceil \ (l - 1)/(k - 1) \rceil \tag{2.6}$$

However, the expansion of the receptive field is not infinite. From the above equation, it can be found that the number of network layers that satisfy the full history coverage is proportional to the input length with a fixed kernel size. This can lead to the network becoming too deep, resulting in a surge in the number of parameters. Also, it has been demonstrated that using a large number of layers leads to the problem of gradient degradation. Therefore, dilation is introduced in TCN to avoid too deep layers.

Dilation refers to the distance between convolution kernel elements in a convolutional layer. Thus, a conventional convolutional layer can be regarded as a 1-dilated layer. The following Fig. 2.14 shows a 2-dilated layer with an input length of 4 and a kernel size of 3. Where padding is done on the left side of the elements.

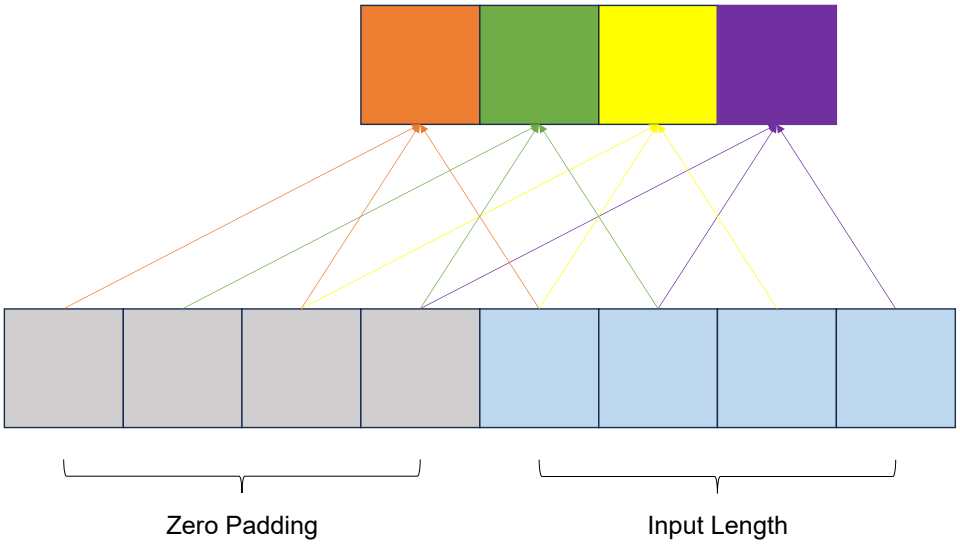


Figure 2.14: An example of 2-dilated layer with input_size of 5 and kernel_size of 3

The receptive field of a 2-dilated layer is increased compared to the previous 1-dilated layer. Indeed, a d-dilated layer with a kernel size k has a receptive field spreading across a length of $1 + d \times (k - 1)$. This does not solve the problem because the number of layers satisfying full coverage is still positively correlated with the input length.

This does not solve the problem because the number of layers satisfying full coverage is still positively correlated with the input length. The network can exponentially increase dilation between layers to reduce the number of layers. This introduces a constant b as dilation_base, which is related to dilation d by $d = b^i$, where i denotes the number of layers. The Fig. 2.13 shows a network with an input length of 10, a kernel size of 3, and a dilation base of 2. Full coverage is satisfied by using 3 dilated convolutional layers. As a comparison, a traditional convolutional network requires 5 layers.

The width of the receptive field w of a TCN with exponential dilation base b, kernel size k, and number of layers n is given by

$$w = 1 + \sum_{i=0}^{n-1} (k-1) \cdot b^i = 1 + (k-1) \cdot \frac{b^n - 1}{b - 1} \tag{2.7}$$

To avoid holes in the receptive field, the receptive field width must be greater than or equal to the input length for full history coverage. i.e.

$$1 + (k-1) \cdot \frac{b^n - 1}{b - 1} \geq l \tag{2.8}$$

Similarly, the minimum number of network layers to satisfy full coverage can be calculated by the following equation:

$$n = \left\lceil \log_b \left(\frac{(l-1)(b-1)}{k-1} \right) + 1 \right\rceil \tag{2.9}$$

Residual Connections The residual blocks (originally from ResNet [63]) allow each layer to learn modifications to the identity mapping and work well in very deep networks. Residual connectivity is important to ensure a valid history of use over time. For example, if a prediction depends on a history length of the 12th power of 2, 12 layers are needed to handle such a large acceptance domain. Each residual block has two layers of dilated causal convolution, normalization of the weights, ReLU activation, and dropout. Suppose the number of input channels differs from the number of output channels of the dilated causal convolution (number of filters of the second dilated convolution). In that case, there is an optional 1x1 convolution. It ensures that the residual join (summing the elements of the convolution output and input) is valid.

An example of the residual block is as follows:

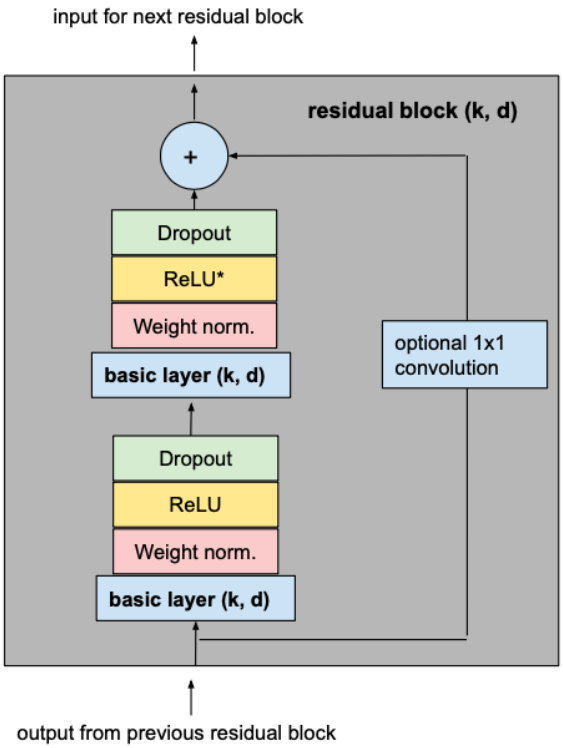


Figure 2.15: An example of a residual block in TCN network (Adapted from [36])

Final design of TCN network The following picture shows the final TCN model with input length of l, kernel size of k, dilation base of b, $k \geq b$, and with a minimum number of residual blocks for full history coverage n, where n can be computed from the other values as explained above.

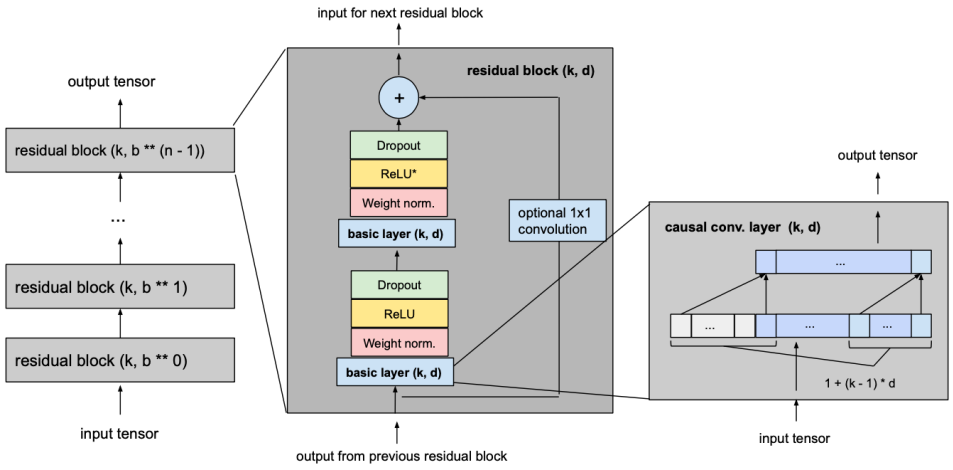


Figure 2.16: The final design of TCN network (Adapted from [36])

2.7.1. Benefits of TCN Model

Compared to regular RNNs(LSTM and GRU), using TCN has the following benefits:

- Unlike RNNs, TCNs can take advantage of parallelism by performing convolution in parallel.

- Users can adjust the perceptual field size by the number of layers, expansion factor, and filter size, which can control the memory size of the model for different domain requirements.
- Unlike RNNs, the gradient is not in the time direction but in the network depth direction, which can make a big difference, especially when the input length is long. As a result, the gradient in TCNs is more stable (also thanks to the residual connectivity).
- The memory requirement is lower than LSTM and GRU because only one filter per layer exists. In other words, the total number of filters depends on the number of layers (not the input length).

2.8. Analysis of Computational Cost

Computational cost serves as a straightforward metric to quantify the resources a neural network consumes during training or inference. It provides a quick assessment of the time or computing power required for network training. This metric can be quantified in various ways, with common approaches including time and the number of computations. The latter is often expressed as either the number of floating-point operations (FLOPs) or the number of multiply-and-accumulate operations (MACs).

2.8.1. RNN

The computation of an RNN involves looping operations, so the computation at each time step needs to be considered. Below is a simple example of an RNN containing the computation process in one time step: Assume an RNN has an input sequence of length T, input dimension D, hidden layer dimension H, and output dimension D.

1. Define the weight matrix:
   - Input data X_t with dimensions $(1, D)$.
   - First hidden layer state h_{t1} with dimension $(1, H)$.
   - Input to the first hidden layer weight matrix W_{xh1} with dimensions (D, H).
   - Weight matrix W_{hh1} from the first hidden layer to the first hidden layer with dimensions (H, H).
   - The bias matrix b_{h1} of the first hidden layer with dimensions $(1, H)$.
   - Second hidden layer state h_{t2} with dimensions $(1, H)$.
   - The weight matrix W_{xh2} input to the second hidden layer with dimensions (H, H).
   - The weight matrix W_{hh2} of the first hidden layer to the second hidden layer with dimensions (H, H).
   - The bias matrix b_{h2} of the second hidden layer with dimensions $(1, H)$.
   - The weight matrix W_{hy} of the second hidden layer to the output layer with dimension (H, D_{out}).
   - Output Y_t with dimensions $(1, D_{out})$.
2. Compute at each time step t:
   - First hidden layer input:

$$h_{t1\_input} = X_t \times W_{xh1} + h_{t1-1} \times W_{hh1} + b_{h1} \tag{2.10}$$

   - First hidden layer state:

$$h_{t1} = h_{t1\_input} \tag{2.11}$$

   - Second hidden layer input:

$$h_{t2\_input} = h_{t1} \times W_{xh2} + h_{t2-1} \times W_{hh2} + b_{h2} \tag{2.12}$$

   - Second hidden layer state:

$$h_{t2} = h_{t2\_input} \tag{2.13}$$

   - Output layer:

$$Y_t = y_{t\_input} = h_{t2} \times W_{hy} \tag{2.14}$$

3. Calculate FLOPs and MACs:

$$FLOPs_{per\_step} = 2 \times (D \times H) + 2 \times H + 2 \times (H \times H) + H \times D \tag{2.15}$$
$$FLOPs_{total} = T \times FLOPs_{per\_step} \tag{2.16}$$
$$MACs_{per\_step} = D \times H + H \times H + H \times H + H \times H + D \times H + H \times H + H \times D \tag{2.17}$$
$$MACs_{total} = T \times MACs_{per\_step} \tag{2.18}$$

2.8.2. CNN

To evaluate the performance of a CNN, in addition to its performance metrics (accuracy in the classification task, error in the estimation task, precision in the detection task, etc.), the model complexity of the CNN, such as the number of parameters and the computational effort, needs to be taken into account. The parameters of a CNN include the convolutional kernel weights, fully-connected layer weights, and other weights to be learned. The number of parameters of CNN is the sum of all these parameters. Since the number of parameters is relatively large, it is usually expressed in M or G. For example, the number of parameters of ResNet50 is 25.56M [63]. The computation of CNN mainly comes from the multiply-and-accumulate (MAC) computation that needs to be performed by the CNN forward inference, so the amount of computation is often used as the number of MACs, which is usually in the order of M or G, and the amount of computation of ResNet50 mentioned above is 4.14G MAC. Since multiply-add calculations are implemented through floating-point operations, floating-point operands can also represent the computation amount. Floating-point operation numbers are often abbreviated as FLOPs.

Convolutional layer parameter calculation

The parameters, or weights, of the CNN convolutional layer are divided into two types: W and b. W denotes a matrix, and also, therefore, it contains more information compared to b and is also the main part of the parameters.

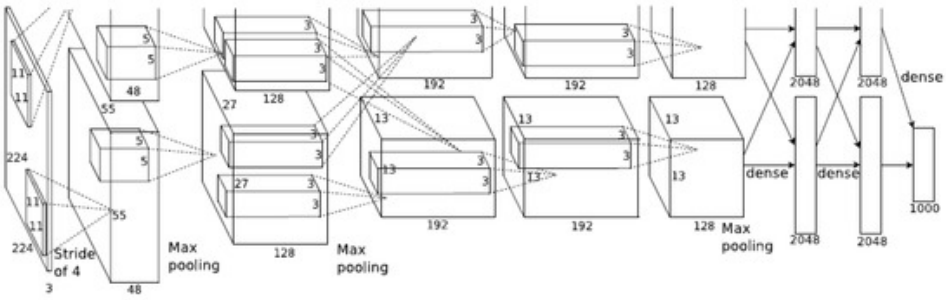


Figure 2.17: The original AlexNet structure[64]

Take the above Fig. 2.17 is an example. In AlexNet, The small rectangle in each large rectangle is weight W; it is a three-dimensional matrix of $[K_h, K_w, C_{in}]$, which K_h is height of the convolution kernel (filter or kernel), K_w is the width of the convolution kernel and C_{in} is number of previous input channels (Channels). Generally, K_h and K_w are of the same size. A convolutional kernel sweeps from left to right and from top to bottom on the previous layer's feature map, then it computes many forward-propagated values, which will be put together into a new feature map according to their original relative positions, with height and width as H_{out} and W_{out}, respectively. Of course, one convolutional kernel extracts too much information, and so it needs different convolutional kernels to sweep the data, which will result in an N-dimensional feature map, i.e., the number of output channels of the current layer $C_{out} = N$. This is summarized as follows: a filter of size $[K_h, K_w, C_{in}]$ crosses a feature map of size $[H_{in}, W_{in}, C_{in}]$ at the previous level and eventually generates a new feature map of size $[H_{out}, W_{out}, C_{out}]$. The process is shown in the following Fig. 2.18.

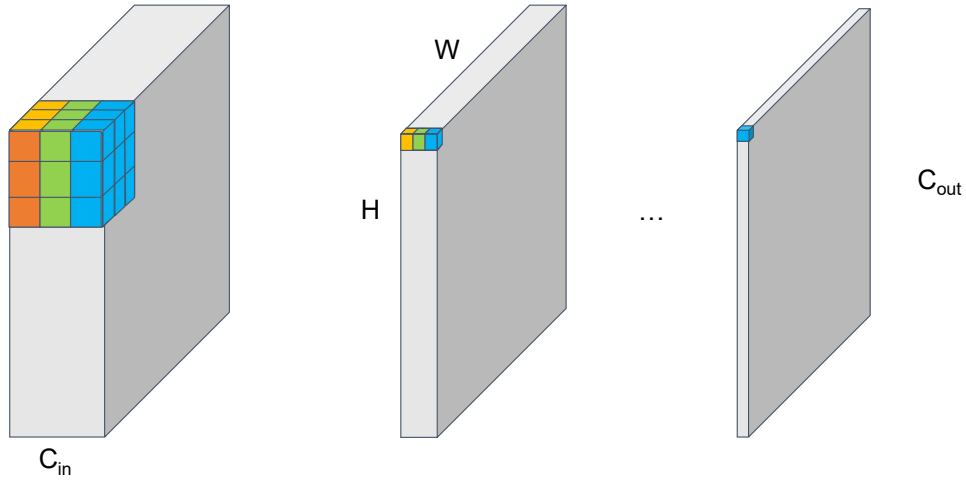


Figure 2.18: Example of how to calculate parameters of convolutional layer

For a given convolutional layer, the number of its parameters, i.e., the sum of the number of weights of W and b, is: $(K_h \times K_W \times C_{in}) \times C_{out} + C_{out}$

Fully-connected layer parameter calculation
The parameter number calculation is simpler for fully-connected layers, such as the last three layers of AlexNet, because it's a multiplication of two sets of one-dimensional data and then adds a bias. If it's a convolutional layer overloaded to a fully-connected layer, such as layer 5 to layer 6 in Fig. 2.17, will first flatten the 3D data of layer 5 to one-dimensional and note that the total number of elements remains unchanged. For a given fully connected layer, if the input data has N_{in} nodes and the output data has N_{out} nodes, it has the number of parameters: $N_{in} \times N_{out} + N_{out}$. If the upper layer is a convolutional layer, N_{in} is the number of elements of the convolutional layer's output 3D matrix: $N_{in} = H_{in} \times W_{in} \times C_{in}$.

FLOPs
Concerning the convolutional layer, the primary floating-point operations involve multiplications related to the filter weights (W) and additions related to the biases (b). Specifically, each element in W corresponds to a multiplication operation, and each element in b corresponds to an addition operation. At first glance, it might seem that the number of FLOPs and parameters are identical. However, one crucial aspect has been overlooked—the values in each layer of the feature map are processed by the same filter, indicating that the weights are shared. This characteristic, intrinsic to CNNs, plays a pivotal role in significantly reducing the number of parameters involved.

Therefore, to calculate the FLOPS, it only need to multiply the parameters by the size of the feature map, i.e., for a certain convolutional layer, the number of FLOPs is:

$$\begin{aligned} num_{para} \times size_{outfeaturemap} &= [(K_h \times K_W) \times C_{in} + 1] \times [(H_{out} \times W_{out}) \times C_{out}] \\ &= [(K_h \times K_W) \times C_{out} + C_{out}] \times [H_{out} \times W_{out}] \end{aligned} \tag{2.19}$$

As for fully-connected layers, Since there is no weight sharing, the number of FLOPs is the number of parameters in that layer: $N_{in} \times N_{out} + N_{out}$.

MACs
The dot product of two vectors with a size of n involves n MACs. In terms of FLOPs, a dot product executes 2n - 1 FLOPs, given the presence of n multiplications and n - 1 additions. Therefore, it can be approximated that one MAC operation is roughly equivalent to two FLOPs.

In a fully-connected layer, every input is connected to every output. In the context of a layer with I input values and J output values, the corresponding weights W can be organized and stored in a matrix of dimensions I × J. The computation performed by a fully-connected layer is:

$$y = matmul(x, W) + b \tag{2.20}$$

In this context, x represents a vector of I input values, W denotes the $I \times J$ matrix containing the layer's weights, and b is a vector of J bias values that are added to the computation. The resulting vector y comprises the output values computed by the layer and has a size of J in the case of a fully-connected layer that undergoes the matrix multiplication operation matmul(x, W). This matrix multiplication is essentially a series of dot products, where each dot product involves the input x and one column in the matrix W. Both x and the columns in W have I elements, constituting I MAC operations. To obtain the final vector y, J of these dot products need to be computed, resulting in a total of $I \times J$ MACs, which is equivalent to the size of the weight matrix. It's worth noting that the bias b doesn't impact the number of MACs. In general, multiplying a vector of length I with an $I \times J$ matrix to produce a vector of length J involves $I \times J$ MACs or ($2I$ - 1) $\times J$ FLOPs.

The input and output to convolutional layers are not vectors but three-dimensional feature maps of size $H \times W \times C$ where H is the height of the feature map, W the width, and C the number of channels at each location. Most convolutional layers used today have square kernels. For a conv layer with kernel size K, the number of MACs is:

$$K \times K \times C_{in} \times H_{out} \times W_{out} \times C_{out} \tag{2.21}$$

Where the bias and the activation function are ignored.

2.8.3. TCN

Calculating FLOPs and MACs in TCNs is similar to calculating CNNs; the key idea is considering the convolution layers. Below is an example of how FLOPs and MACs are calculated in a TCN model. Assume that the TCN contains two convolutional layers, a ReLU activation function, and a maximum pooling layer.

1. Define the weight matrix:
   - Weight matrix W_1 of convolution kernel 1 with dimensions $(D_{in}, D_{hidden1}, K_1)$.
   - Weight matrix W_2 of convolution kernel 2 with dimensions $D_{hidden1}, D_{out}, K_2)$.
   - The bias term b_1 of the first convolutional layer, of dimension $(D_{hidden1},)$.
   - The bias term b_2 of the second convolutional layer, of dimension $(D_{out},)$.
2. Compute at each time step t:
   - Input data X_t with dimensions $(1, D_{in}, L)$.
   - Output Y_t with dimensions $(1, D_{out}, L)$.
3. Computational process:
   - First convolutional layer:
   $$Z_1 = conv(X_t, W_1) + b_1 \tag{2.22}$$
   - Nonlinear activation function (ReLU):
   $$A_1 = ReLU(Z_1) \tag{2.23}$$
   - Second convolutional layer:
   $$Z_2 = conv(A_1, W_2) + b_2 \tag{2.24}$$
   - Nonlinear activation function (ReLU):
   $$A_2 = ReLU(Z_2) \tag{2.25}$$
   - Maximum Pooling:
   $$Y_t = maxpool(A_2) \tag{2.26}$$

4. Calculate FLOPs and MACs:

$$FLOPs_{per\_step} = D_{in} \times D_{hidden1} \times K_1 \times L + D_{hidden1} \times D_{out} \times K_2 \times L \quad (2.27)$$
$$FLOPs_{t}otal = T \times FLOPs_{per\_step} \quad (2.28)$$
$$MACs_{conv1} = D_{in} \times D_{hidden1} \times K_1 \times L \quad (2.29)$$
$$MACs_{conv2} = D_{hidden1} \times D_{out} \times K_2 \times L \quad (2.30)$$
$$MACs_{per\_step} = MACs_{conv1} + MACs_{conv2} \quad (2.31)$$
$$MACs_{total} = T \times MACs_{per\_step} \quad (2.32)$$

2.9. Model Compression Methods

2.9.1. Quantization

A neural network has interconnected neurons organized into layers. In Fig. 2.19, a typical neural network is composed of layers of neurons, each equipped with its unique weights, biases, and associated activation functions.

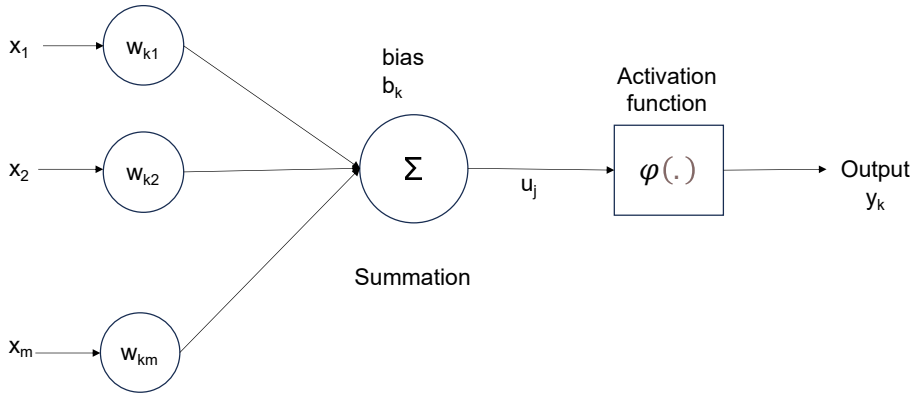


Figure 2.19: An example of a neuron within a neural network.

The ”parameters” of a neural network, referring to its weights and biases, play a crucial role in its functioning. Additionally, each neuron possesses its unique ”activation,” determining its level of activity. The activation is influenced by the weights, biases, and the chosen activation function. During training, adjustments are made to these parameters, consequently affecting the neuron’s activation. In terms of memory storage, the neural network mainly retains values for weights, biases, and activations. Typically represented as 32-bit floating-point values, this format ensures high precision, enhancing the network’s accuracy. However, the precision comes at the cost of increased memory usage, particularly for networks with millions of parameters and activations. For instance, the 50-layer ResNet architecture encompasses approximately 26 million weights and 16 million activations. Using 32-bit floating-point values for both weights and activations would necessitate 168 MB of storage for the entire architecture.

Quantization involves decreasing the precision of weights, biases, and activation values to minimize memory consumption. Simply, this process transforms the 32-bit floating-point representation of neural network parameters into a more compact form, such as an 8-bit integer. Transitioning from a 32-bit to an 8-bit representation results in a fourfold reduction in model size, showcasing a notable advantage of quantization – substantial memory reduction. Fig. 2.20 shows an example of quantization.

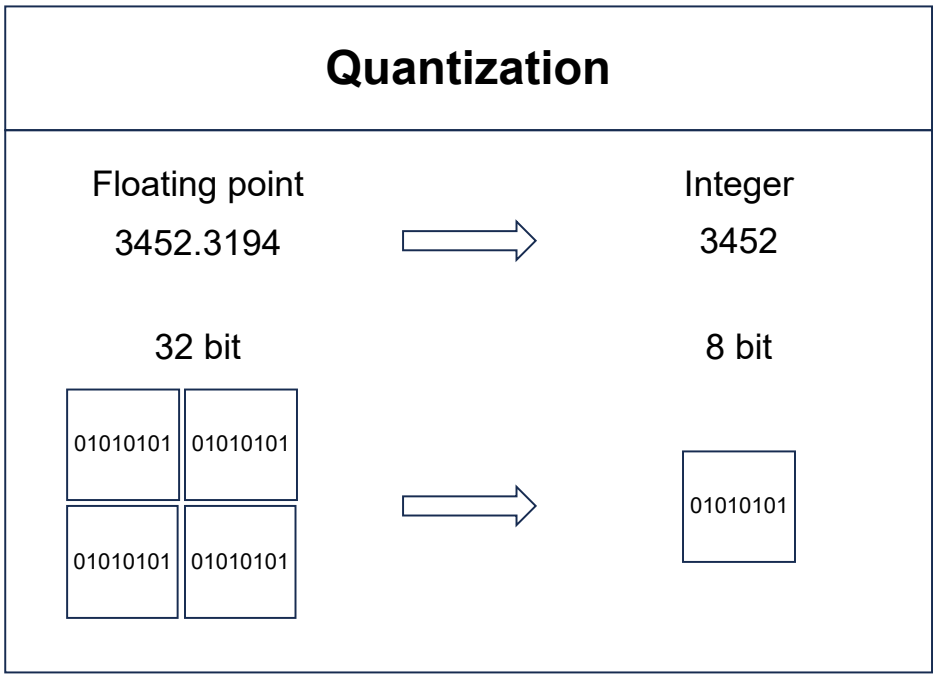


Figure 2.20: Example of how quantization shrinks neural networks

Quantization offers an additional advantage by reducing network latency and enhancing energy efficiency. The improvement in network speed comes from the use of integer operations instead of floating-point data types. These integer operations generally demand less computation across various processor cores, including microcontrollers. The overall gain in power efficiency is attributed to a reduction in computations and memory accesses.

While the benefits of quantization are substantial, it comes at the expense of potential loss of accuracy in neural networks, as they may not precisely represent information. However, the degree of accuracy loss, the specific network architecture, and the training/quantization scheme all influence the impact. Typically, quantization results in only a minimal accuracy loss, especially when compared to the notable improvements in latency, memory usage, and power efficiency.

In practice, there are two main methods of quantification:

- Post-training quantization
- Quantization-aware training

Post-training quantization is a method where a neural network undergoes training exclusively using floating-point computation and is subsequently quantized. Following the completion of training, the neural network is frozen, and its parameters unchangeable. Subsequently, the parameters will be quantized. The resultant quantized model is then deployed for inference without further alterations to the post-training parameters. Despite its simplicity, this approach may incur a higher loss of accuracy. This is attributed to the fact that all errors related to quantization manifest after the completion of training, making it challenging to compensate for these errors.

Quantization-aware training addresses quantization-related errors by incorporating the quantized version into the forward pass during the training process. The concept behind this approach is that quantization-associated errors accumulate within the overall loss of the model during training. Subsequently, the training optimizer adjusts the parameters accordingly, mitigating the overall error. The key advantage of quantization-aware training lies in the significantly reduced quantization loss compared to the post-training quantization approach.

Quantization has been discussed for a long time. It was introduced in the early work of weight binarization [65] and model compression [66]. In the early stages of research, it was common to enforce the

same accuracy requirements for the weights of different network layers, favoring the reduction of the value [67] and distribution [66] differences between the quantized weights and the full accuracy weights. [68] introduces a quantization method based on learning where the quantizer undergoes training using data. Regularizer for quantization is also proposed to implement binarized weights [69]. Research has also explored the aggregation of multiple models with reduced precision, as presented in [70]. This research illustrates enhanced performance compared to a single model within equivalent computational constraints.

2.9.2. Sparsity

Various machine learning models suffer from a very fatal problem: the huge amount of computation and access memory of neural networks, which puts high demands on performance (energy consumption, latency, access memory, etc.). One possible solution is to utilize sparsity in machine learning models fully. Sparsity refers to a numerical matrix containing many zeros or values that do not significantly affect the computation so that computational processes involving these zero elements can be skipped to save computational resources and accelerate the training of the neural network. Reducing the amount of computation can ensure that unnecessary storage and computation are reduced, improving the performance of the machine learning model when it is deployed.

Activation Sparsity

In DNNs, each layer's output (activations) is typically dense, with 50% to 100% of neurons having non-zero activations. Although less discussed, activation sparsity can also be applied to DNNs. in terms of activation sparsity, determining the neurons to be activated is often done by explicitly selecting the first k activations or by calculating a dataset-specific activation threshold for the neurons, which reduces, on average, the number of activated neurons to a desired level. In addition, the overall magnitude of the activations can be reduced by introducing appropriate regularizers, thereby eliminating large activation values.

Weight Sparsity

Weight sparsity has consistently been leveraged in weight pruning. Weight sparsity is a well-explored algorithm for network sparsification that eliminates redundant connections within the network. The inception of weight pruning dates back to [71], where network weights were pruned based on the second derivative of the loss function. Recent investigations, such as [72] and [73], have demonstrated the effectiveness of pruning in compressing deeper networks with minimal impact on accuracy.

Weight pruning methods have weaknesses across three key aspects: pruning granularity, criteria, and recovery strategies. Fine-grained pruning at the element level introduces irregular memory access and computation patterns, presenting challenges for efficient utilization on accelerators. Alternatively, studies such as [74] have explored pruning with coarse granularity to facilitate easier exploitation of sparsity. Fig. 2.21 illustrates a spectrum of potential pruning granularities.

Group-level pruning, as proposed in [75], generates zero-valued groups within the kernel, with the group being either a 1-D vector or a pre-defined kernel pattern. Additionally, kernel-level pruning, as discussed in [75] and [76], eliminates unimportant kernels, resulting in reduced input channels while maintaining network density. Filter-level pruning, explored in [77] and [73], introduces extra sparsity at the output activation by eliminating the entire connection to a specific output channel.

In summary, coarse-grained pruning enables more regular data access and easier control, albeit with a larger accuracy drop compared to fine-grained pruning.

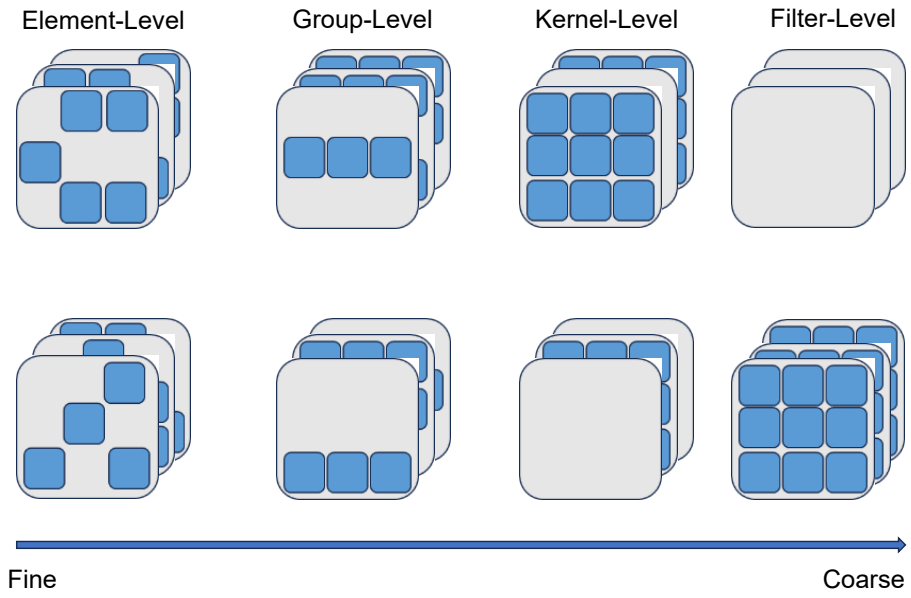


Figure 2.21: Weight pruning with different granularity, redrew from [78]

The goal of the pruning procedure is to categorize pruning. For sparse inference objectives, a pruning pipeline proposed in [72] advocates an iterative fine-tuning approach. In the specific case of AlexNet applied to the ImageNet dataset, the fine-tuning process takes 2.31$\times$ longer than the pre-training phase. To address the challenge of long-time re-training, various pruning methodologies of one-shot fine-tuning, necessitating fine-tuning only once, as demonstrated in [77] and [75]. When the focus is on sparse training, pruning becomes an integral part of the training process right from the beginning.

Table 2.1 provides an overview of pruning methods categorized by their target, granularity, criteria, and re-training approach. Some studies, such as [75] and [76], employ multiple granularity levels simultaneously to enhance efficiency. The discussion covers both inference-targeted pruning methods, involving iterative or one-shot retraining, and training-targeted pruning methods, integrated from the outset.

Particularly is the approach introduced in [76], which incorporates reinforcement learning to automate the determination of target sparsity for each layer. Throughout the network training process, it continually receives accuracy and model complexity as rewards after pruning. Additionally, [79] focuses on reducing the training time of a pruned model by gradually increasing the pruning ratio from scratch until accuracy converges. Another strategy, presented in [80], identifies weights to be pruned by tracking the magnitude of the weight gradient accumulated over time. This approach maintains the target sparsity at a high value, facilitating acceleration during the early pruning stages.

Algorithm	Target	Granularity	Criteria	Re-training
Han et al.[72]	Inference	Element	Magnitude	Iterative
He et al.[81]	Inference	Channel	Optimization	Iterative
Li et al.[77]	Inference	Filter	Magnitude	One-Shot
PatDNN[75]	Inference	Kernel/Group	Optimization	One-Shot
He et al.[73]	Inference	Filter	Geo Median	Iterative
AMC[76]	Training	Element/Kernel	Magnitude	Scratch
Eager Pruning[79]	Training	Element	Magnitude	Scratch
Procrustes[80]	Training	Element	Magnitude	Scratch

Table 2.1: Summary of network pruning methods, adapted from [82]

Temporal Sparsity

In general, when an interval in a segment of a signal remains constant, it can be assumed that there is temporal sparsity in this interval. If a temporal signal is to be processed in a DNN, temporal sparsity can help the network to build sparse matrices and skip the zero computation in them.

3

Proposed Methodology

3.1. Introduction

This section will discuss the two main pruning methods: temporal sparsity and structural weight sparsity. The temporal sparsity is applied to TCN by creating a Delta 1D convolutional layer. The modified TCN is called DeltaTCN. Also, the DeltaTCN has a user-defined threshold to control sparsity during training. Moreover, the structural weight sparsity is applied to TCN by structured weight pruning. The specific model is called Structural Sparse Temporal Convolution Network (SSPTCN). Like DeltaTCN, the sparsity of SSPTCN is also controlled by a user-defined threshold. The threshold can be regarded as a target sparsity; during the training phase, it will gradually increase until it reaches the target sparsity.

3.2. Delta Network Formulation

The idea of a delta network is to transform a dense matrix multiplication into a sparse matrix multiplication. The transformation can drastically reduce the amount of computation and memory access without compromising accuracy.

To illustrate how Delta networks work, assume a matrix multiplication

$$y = Wx \tag{3.1}$$

Where W is the matrix of size $n \times n$, x is the input vector of size n, and y is the output of size n. According to [83], this equation contains n^2 compute operations, $n^2 + n$ read operations, and n write operations. The equation can be applied in a sequence of input $X = \{x_t \mid 1 \leq t \leq T, t \in \mathbb{N}\}$and output $Y = \{y_t \mid 1 \leq t \leq T, t \in \mathbb{N}\}$.

$$y_t = Wx_t \tag{3.2}$$

Now, the above equation can be rewritten into

$$y_t = W\Delta + y_{t-1} \tag{3.3}$$

Where $\Delta = x_t - x_{t-1}$, y_{t-1} is the output of previous time step. If the result of the previous time step y_{t-1} is saved, there will be no compute cost on calculating y_{t-1}, which can be read from the previous state. The result of the Delta network is equal to the original equation; the proof is as follows:

$$\begin{aligned} y_t &= W\Delta + y_{t-1} \\ &= W(x_t - x_{t-1}) + y_{t-1} \\ &= W(x_t - x_{t-1}) + W(x_{t-1} - x_{t-2}) + y_{t-2} \\ &... \\ &= Wx_t \end{aligned} \tag{3.4}$$

3.3. Dense TCN

To design a DeltaTCN, it is necessary to know to which layer of TCN the Delta network algorithm can be applied. Also, it can be helpful to have a comparison between dense TCN and DeltaTCN. This section will start with dense TCN.

According to the paper [36], the most central computation of the TCN network is the one-dimensional convolution (1D-CNN). One-dimensional convolution utilizes multiple convolution kernels of fixed size to perform convolution operations with the input sequence to generate the output sequence. The shape of the convolution kernel is determined by the number of input channels $in\_channels$ and the size of the convolution kernel $kernel\_size$. In contrast, the number of convolution kernels is determined by the number of output channels $out\_channels$.

In the original 1D convolution, the input size is $[N, C_{in}, L_{in}]$ and the output size is $[N, C_{out}, L_{out}]$, where N is the batch size, C is the number of channels and L is the length of the signal sequence. The L_{out} is calculated by the following equation:

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times padding - dilation \times (kernel\_size - 1) - 1}{stride} - 1 \right\rfloor \tag{3.5}$$

The 1D convolution process is shown in Fig. 3.1 and Fig. 3.2 below.

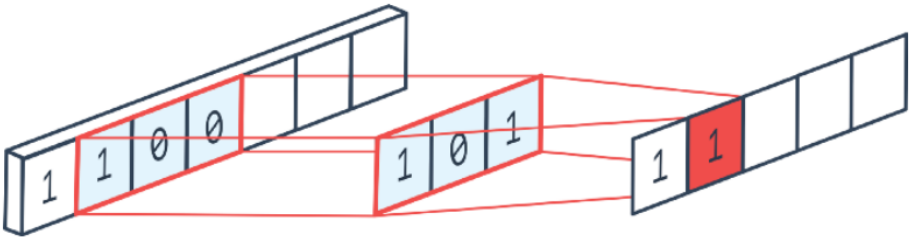


Figure 3.1: 1D convolution example (i)

Figure 3.2: 1D convolution example (ii)

Typically, the 1D-CNN is given as

$$y_t = W * x_t \tag{3.6}$$

This can be easily performed by the command $torch.nn.Conv1d$ in pytorch.

3.4. DeltaTCN

From the previous section, the key layer to create Delta is the 1D convolution layer, and the temporal sparsity will be applied to this layer by the idea of Delta.

As discussed above, 3.6 can be replaced by delta values, forming:

$$\Delta_x = x_t - x_{t-1} \tag{3.7}$$

$$y_t = W\Delta_x + z_x \tag{3.8}$$

Where z_x is defined as the stored result of the previous computation. It can be calculated as follows:

$$z_x = z_{x,t-1} = y_{t-1} = W(x_{t-1} - x_{t-2}) + y_{t-2} \tag{3.9}$$

The initial state of $z_0 = x_0 = 0$.

By definition, DeltaTCN should get the same results as the original TCN model. The mechanism of current DeltaTCN is to skip operation when $\Delta_x = 0$. Since two neighboring inputs will rarely be exactly equal to 0 when calculating Δ in the actual computation process, the sparsity of the original DeltaTCN network is usually not high enough to achieve the purpose of reducing the computation. According to [84], an approximate skipping approach can be applied to the current DeltaTCN model. A threshold is introduced to the DeltaTCN. If $\Delta_x < \Theta$, then $\Delta_x \leftarrow 0$ and the vector-multiplication operation will be skipped. When the threshold is added, the calculation of each update is different from the original TCN, but not by much. However, the addition of the threshold improves the sparsity rate of the model.

If Δ_x is assigned to 0 every time the value of delta is less than the threshold, the error of each computational update will accumulate, resulting in a final result that differs significantly from the TCN. To ensure the accuracy of DeltaTCN, a new intermediate variable $\hat{x}$ is introduced here to record the last changed state.

To be specific, $\hat{x}$ is used as follows:

$$\hat{x_{t-1}} = \begin{cases} x_{t-1} & \text{if } |x_t - \hat{x_{t-1}}| > \Theta \\ \hat{x_{t-2}} & \text{otherwise} \end{cases} \tag{3.10}$$

$$\Delta x_t = \begin{cases} x_t - \hat{x_{t-1}} & \text{if } |x_t - \hat{x_{t-1}}| > \Theta \\ 0 & \text{if } |x_{t-1} - \hat{x_{t-1}}| < \Theta \end{cases} \tag{3.11}$$

This time, the input x_{t-1} from the previous time step is saved as an intermediate state $\hat{x_{t-1}}$. When $\Delta_x > \Theta$, the saved $\hat{x_{t-1}}$ remains unchanged, and when $\Delta_x < \Theta$, the saved $\hat{x_{t-1}}$ will be replaced by an update of x_{t-2}. Therefore, when calculating Δ_x, $\Delta_x = 0$ when $\Delta_x < \Theta$, otherwise $\Delta_x = x_t - \hat{x_{t-1}}$. This operation avoids the accumulation of errors and ensures DeltaTCN's accuracy.

Here is an example of creating a Delta layer. Assume an input sequence of $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix}$, and a kernel of $\begin{bmatrix} 1 & 2 \end{bmatrix}$. For a dense 1D convolution layer, the output should be:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 5 & 8 & 11 & 14 & 17 \end{bmatrix} \tag{3.12}$$

As for a DeltaTCN, the first step is to assume $\Delta x_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}$. Thus, the initial $Y_0 = 0$ at $t = 0$.

For $t = 1$, the Delta calculation is as follows:

$$\Delta x_1 = x_1 - x_0 = \begin{bmatrix} 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \end{bmatrix} \tag{3.13}$$

$$\Delta x_1 * \omega = \begin{bmatrix} 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 \end{bmatrix} = 2 \tag{3.14}$$

$$y_1 = 2 + y_0 = 2 \tag{3.15}$$

For $t = 2$, the Delta calculation is as follows:

$$\Delta x_2 = x_1 - x_1 = \begin{bmatrix} 1 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{3.16}$$

$$\Delta x_2 * \omega = \begin{bmatrix} 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 \end{bmatrix} = 3 \tag{3.17}$$

$$y_2 = 3 + y_1 = 5 \tag{3.18}$$

Based on the above calculation, apply a threshold to Δx during every time step. The threshold is used to control the sparsity. For every element in Δx, if $\Delta x < threshold$, $\Delta x = 0$. For example, if $threshold = 1.5$, the new $\Delta x_2 = \begin{bmatrix} 0 & 0 \end{bmatrix}$. By introducing the $threshold$, the temporal sparsity can be created from the input sequence in the 1D convolution layer.

3.5. SSPTCN

DeltaTCN explores temporal sparsity in the 1D convolution layer and, more specifically, creates Delta elements in the input of 1D convolutions. This kind of sparsity is random sparsity or unstructured

sparsity. The unstructured sparsity may cause extra processing time during training. It's more efficient to generate a structured sparsity. That's the purpose of SSPTCN.

In SSPTCN, a sparse matrix of weights needs to be constructed. To construct the weight sparse matrix, the weights of each layer in the one-dimensional convolution need to be operated. First, extract the weight of each convolutional layer and then perform L2 normalization on the weight matrix. The L2 normalization is calculated as follows:

$$\mathbf{X}_2 = \left(\frac{x_1}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}}, \frac{x_2}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}}, \cdots, \frac{x_n}{\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}} \right) \tag{3.19}$$

Then, sort the L2-normalized weight matrix in descending order. Based on the sorting results, select a final portion of the weight; the exact percentage can be customized and entered. Return the selected weight to 0. Finally, the modified weights are restored according to the original matrix; each element is restored to its original position. The flowcharts of the methodology are given in Fig. 3.3.

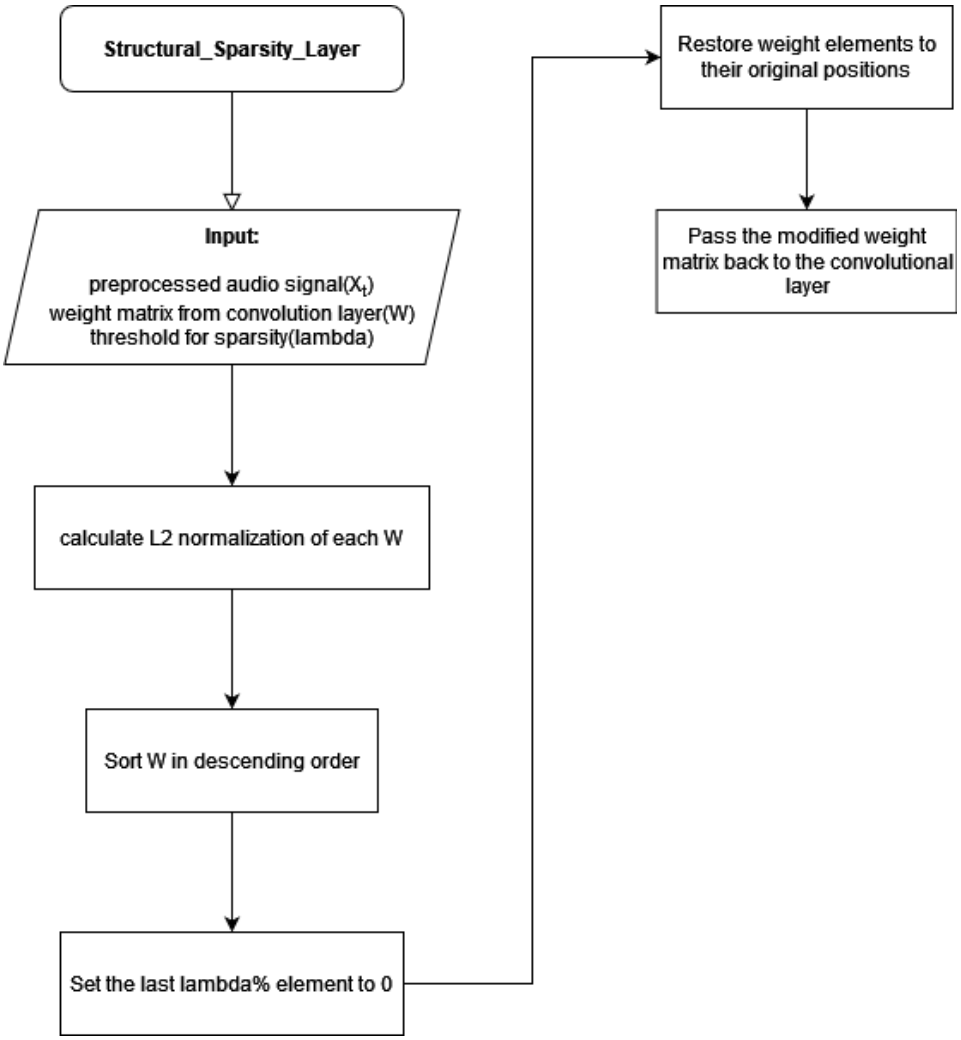


Figure 3.3: Methodology flow of creating a structural sparse weight matrix

The advantage of using structural sparsity is that the weight sparsity matrix is constructed once at the end of each epoch to update the weights constantly. This ensures that the weights of the convolutional layer retain the most important parameters during the training process of each epoch. In addition, although a one-dimensional convolution is used in TCN networks, the batch size is fixed and greater than one at each iteration. Therefore, the de facto convolutional computation can be treated as a two-dimensional convolution.

4

Experiment and Results

4.1. Introduction

This section describes the results of DeltaTCN training as well as retraining. The dataset for training is the Google Speech Commands Dataset V2 [85].

4.2. Experiment Setup

4.2.1. Dataset

The Google Speech Commands Dataset is a collection of over 65,000 audio recordings of people saying 30 different words. It is a dataset researchers and developers use to train and evaluate AI models for speech recognition. This dataset has been used to train models that can recognize spoken commands and understand the context of spoken words. The dataset contains recordings of words like “yes”, “no”, “up”, “down”, “left”, “right”, and more(see Fig. 4.1). The audio clips were recorded in a variety of different environments and with a variety of different accents. The Google Speech Commands Dataset can be used to train AI models for various applications. For example, it can be used to train models for voice-controlled devices such as Amazon Alexa or Google Home. It can also be used to train models for voice recognition, voice search, and voice biometrics. Additionally, this dataset can be used to train models for natural language processing, which is used in applications such as chatbots, virtual assistants, and more. This dataset has been used to train models for various applications, such as voice-controlled devices, voice recognition, voice search, voice biometrics, and natural language processing.

yes	no	up	down	left	Keyword
right	on	off	stop	go	
zero	one	two	three	four	Non-Keyword
five	six	seven	eight	nine	
bed	bird	cat	dog	happy	
house	Marvin	Sheila	tree	wow	
backward	forward	follow	learn	visual	

Figure 4.1: List of the words included in the Google Speech Commands V2[86]

4.2.2. Preprocess

Log Mel-filterbank energy features (LFBE) are commonly used for speech feature extraction in KWS tasks. LFBE arranges a set of bandpass filters from low to high frequencies in a frequency band from

dense to sparse according to the size of the critical bandwidth to filter the input signal. The signal energy output from each band-pass filter is used as the basic feature of the signal, which can be used as the input feature of speech after further processing. Since this feature does not depend on the nature of the signal, it does not make any assumptions or restrictions on the input signal and again utilizes the findings of the auditory model. As a result, this parameter has better robustness, is more in line with the auditory properties of the human ear, and still has good recognition performance when the signal-to-noise ratio is reduced.

The basic process of LFBE extraction is as follows:

1. Pre-emphasis

   Passes the speech signal through a high-pass filter. The purpose of pre-emphasis is to boost the high-frequency portion of the signal so that the spectrum of the signal is flattened, keeping it in the entire frequency band from low to high frequencies and being able to use the same signal-to-noise ratio for the spectrum.

   $$s_n^{'} = s_n - k \cdot s_{n-1} \tag{4.1}$$

2. Hamming Window

   Window addition is used to smooth the signal. Smoothing with a Hamming window attenuates the size of the side flaps and spectral leakage after the FFT compared to the rectangular window function.

   $$s_n^{'} = \left\{ 0.54 - 0.46 \cos(\frac{2\pi(n-1)}{N-1}) \right\} \cdot s_n \tag{4.2}$$

3. Frequency domain conversion

   Fast Fourier transform is performed on each frame of the windowed signal to obtain the spectrum of each frame. The power spectrum of the speech signal is obtained by taking the mode square of the spectrum of the speech signal.

   Amplitude spectrum:

   $$S_i(k) = \sum_{n=1}^{N} s_i(n) e^{-j2\pi kn/N} 1 \leq k \leq K \tag{4.3}$$

   Power spectrum:

   $$P_i(k) = \frac{1}{N} \left| S_i(k)^2 \right| \tag{4.4}$$

4. Filtering with Mel Scale Filter Banks

   Because there is a lot of redundancy in the frequency domain signals, the filter bank can streamline the magnitude of the frequency domain by representing each frequency band with a single value.

5. Use Log to state energy values

   Since the perception of sound by the human ear is not linear, it is better described by a nonlinear relationship such as a log. The cepstrum analysis can only be done after the log is taken.

Fig. 4.2 shows an example of a ”backward” raw signal from the Google Speech Commands dataset in the time domain. Fig. 4.3 shows the Filter bank on a Mel-Scale and Fig. 4.4 shows the final log filter bank features of a ”backward” signal.

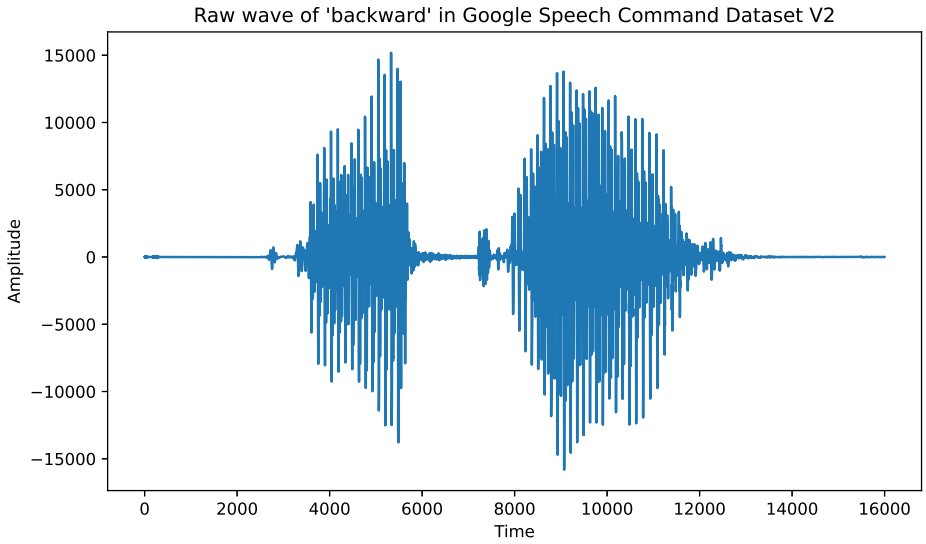


Figure 4.2: Example spectrum of ”backward” signal in time domain

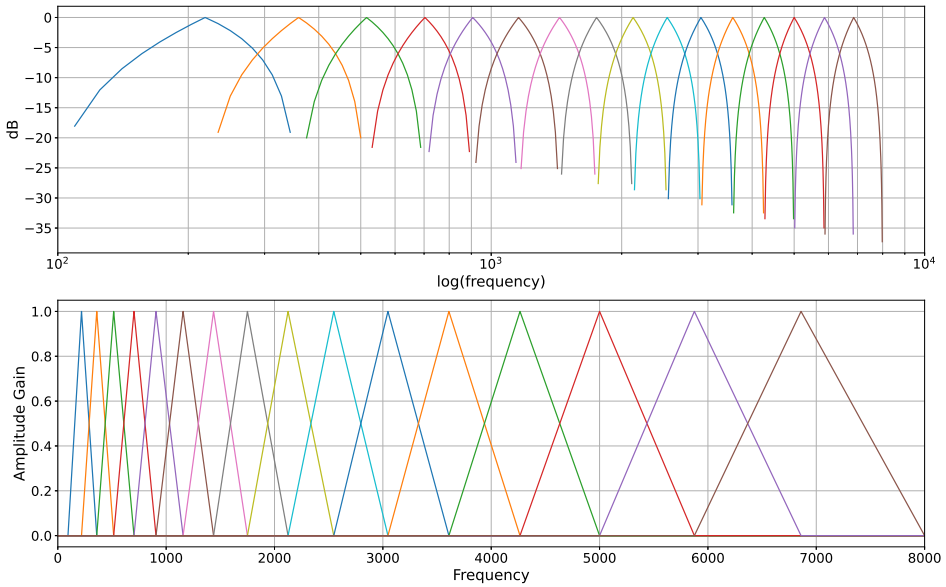


Figure 4.3: Filter bank of ”backward” on a Mel-Scale

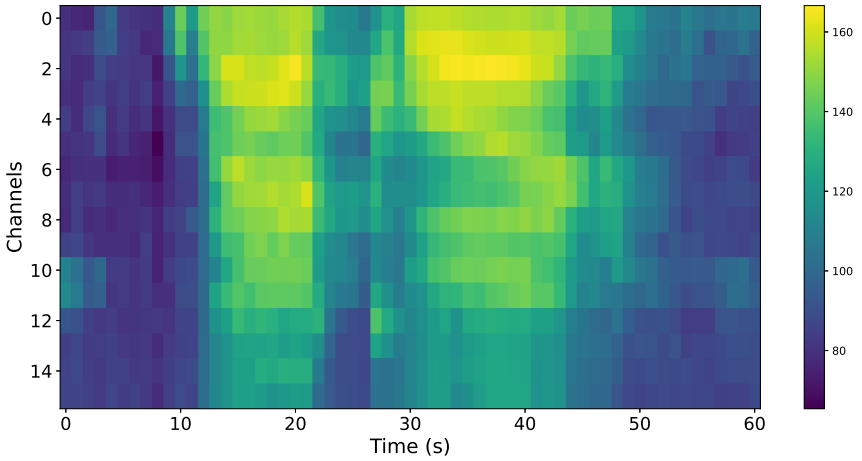


Figure 4.4: log filter bank features of ”backward” speech signal

4.2.3. Neural Network Training

Baseline model training

First, an ordinary TCN model must be trained to explore its performance on the KWS task and use the results as a baseline value. A standard TCN network, as well as a DeltaTCN network, was trained. For the TCN network, kernel size and stride are constant values of 3 and 1. The input size is fixed at 16. The number of channels is defined by multiplying TCN channels and TCN layers. In this project, the TCN channels are increased from 16 to 256, and the TCN layers are increased from 1 to 6. Use a loop to iterate through each combination of channels and layers and derive the number of parameters and the accuracy under that number of channels. Each combination of channel numbers is training for 30 epochs. The setup parameters can be seen in Table4.1.

The results in this part serve only as baseline values, so no sparsification is involved, and its associated data are not examined in the subsequent discussion.

Building DeltaTCN

To explore the effect of sparsity on TCN networks, the experiments in the second part sparsify the inputs to the convolutional layer, i.e., constructing DeltaTCN networks. For the DeltaTCN network, choose the best-performing model generated from the trained TCN network and retain it under the same Google Speech Command dataset. The parameters are set to the same as those of the best-performing TCN network. Besides, add a new delta threshold parameter to realize the DeltaTCN network. In this part, the threshold is varied from 0 to 0.5. Similarly, record the accuracy of the retrained DeltaTCN network and the sparsity. The setup parameters of the DeltaTCN network are listed in table 4.1.

Building SSPTCN

The third part of the experiment is still based on the TCN network of the first part. The difference this time is to utilize structured sparsity. In this part of the experiment, the weights of the convolutional layers will be constructively sparsified, i.e., SSPTCN. For SSPTCN, all training is based on the number of input channels at 64. There are two variables: one is the kernel size, and the other is the number of input layers. The third part of the experiment will traverse core sizes 3, 5, and 7, as well as traverse input layers 4 and 5. Also, SSPTCN has to introduce a custom parameter threshold. this parameter is used to determine the sparsity of the weights of the convolutional layers, and the detailed principle is introduced in Chapter 3. The parameters used in part 3 are shown in Table4.2.

Since the third part is to sparsify the weights, the epoch is increased to 100 to ensure the accuracy of the training. the threshold will be gradually increased to the target parameter in the first 50 rounds of training, and the target threshold will be maintained in the last 50 rounds to continue the training. For example, suppose the target threshold for this training is 50%. Then, in the first 50 training rounds, the initial input threshold will be 0. After each round of training, the threshold will be increased by 1% until

the target threshold of 50% is reached at the end of the 50th round of training. Then, the threshold will be kept set at 50% for the next 50 training.

	TCN	DeltaTCN
Kernel Size	3	3
Stride	1	1
Number Of Channels	16, 32, 64, 128, 256	64
Number Of Layers	1, 2, 3, 4, 5, 6	5
Delta Threshold	NA	0, 0.1, 0.2, 0.3, 0.4, 0.5
Epoch	30	10

Table 4.1: Network parameters

Model	**SSPTCN**
Kernel Size	3, 5, 7
Stride	1
Input Channel	64
Input Layer	4, 5
Threshold	0-0.9
Epoch	100

Table 4.2: Parameters of SSPTCN

4.3. Results

4.3.1. Baseline - TCN

The results are listed in table 4.3 by scanning parameters such as channel number and layer number of the TCN network. As the number of input channels and layers increases, the number of parameters in the TCN network inevitably increases, and the number of input channels is the main influencing factor. Of course, the accuracy of network training increases with the increase of parameter two. Contrary to the number of parameters, the accuracy of training is mainly affected by the number of network layers, and the training accuracy increases dramatically with each additional layer of the network until saturation.

Channel	Layer	Parameter	Accuracy
16	1	1804	20.386%
16	2	3404	28.759%
16	3	5004	48.590%
16	4	6604	80.569%
16	5	8204	82.700%
16	6	9804	84.299%
32	1	5676	22.874%
32	2	11948	34.554%
32	3	18220	60.004%
32	4	24492	90.318%
32	5	30764	90.851%
32	6	37036	91.872%
64	1	17484	24.384%
64	2	42316	36.731%
64	3	67148	67.200%
64	4	91980	93.271%
64	5	116812	94.115%
64	6	141644	94.204%
128	1	59532	25.561%
128	2	158348	40.151%
128	3	257164	70.153%
128	4	355980	94.270%
128	5	454796	95.026%
128	6	553612	95.314%
256	1	217356	27.049%
256	2	611596	41.062%
256	3	1005836	72.951%
256	4	1400076	95.159%
256	5	1794316	95.314%
256	6	2188556	95.403%

Table 4.3: Accuracy and model size results of the baseline TCN network

Generally, the maximum number of parameters leads to the highest training accuracy. However, it has been observed that if the number of input channels or the number of network layers is appropriately reduced, there is no significant loss in accuracy, but a significant reduction in the number of parameters can be obtained. For example, the group Channel=256, Layer=6 has an accuracy of 95.403%. When the Channel is reduced to 128 and the Layer remains the same, the accuracy is 95.314%, a mere 0.089% reduction. However, the number of parameters decreases from 2188556 to 553612, which is a 74.7% decrease in the number of parameters. This is a huge reduction in both computation and memory throughput.

By comparing the accuracy and the number of parameters, the number of input channels 64 and 128 were selected as the reference group for the TCN model in this project. By comparing the accuracy and the number of parameters, the number of input channels 64 and 128 were selected as the reference group for the TCN model in this project. Their respective relationships of parameters and accuracy at different numbers of input layers are plotted. See Fig. 4.5 and Fig. 4.6.

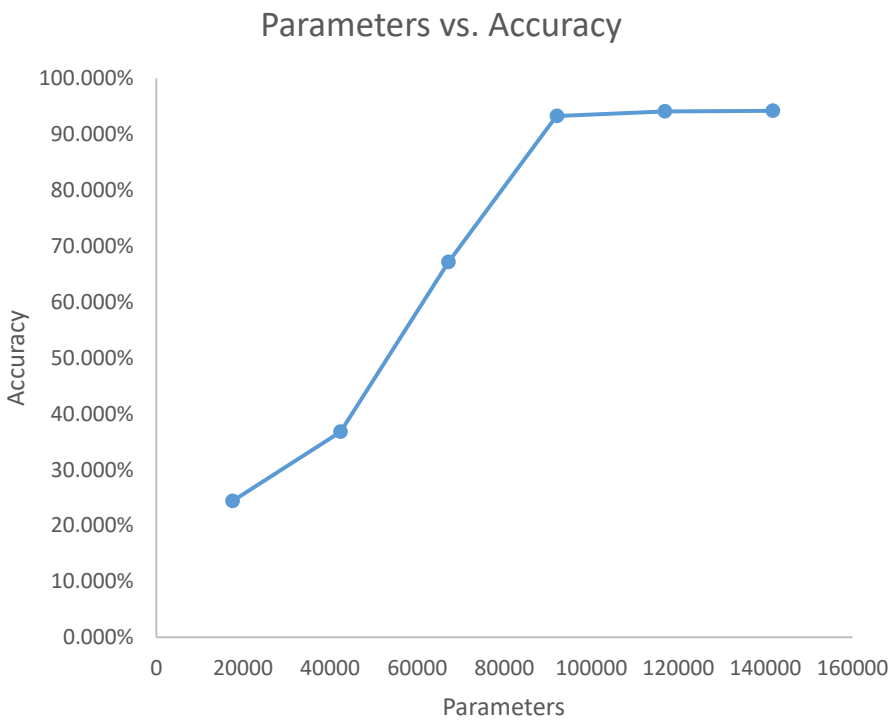


Figure 4.5: Parameters vs Accuracy(TCN model of 64 input channels and input layers sweep from 1 to 6

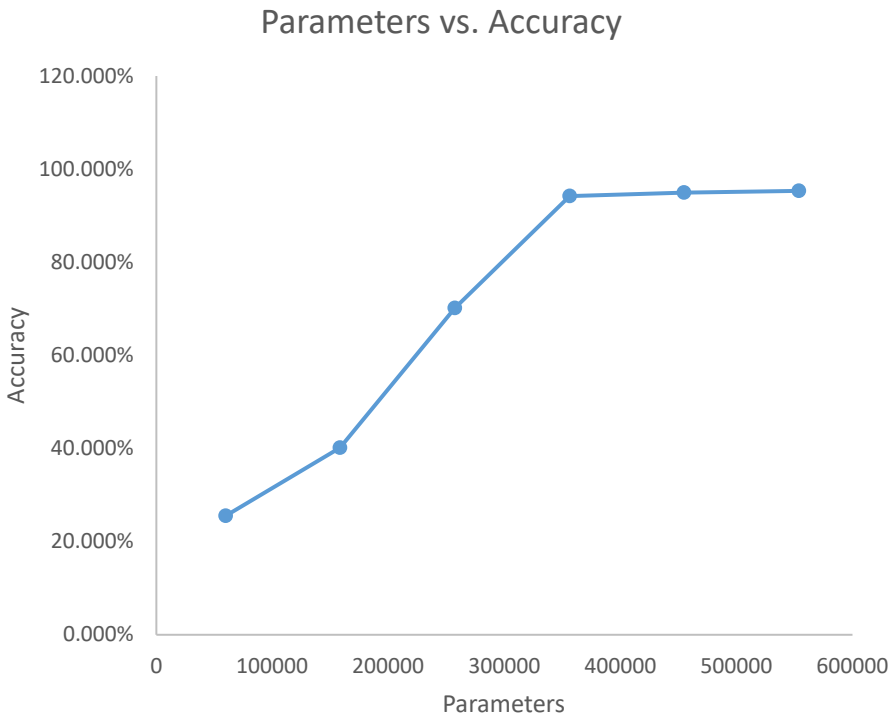


Figure 4.6: Parameters vs Accuracy(TCN model of 128 input channels and input layers sweep from 1 to 6)

The graph of the relationship between the number of parameters and accuracy shows that the number

of parameters can affect the accuracy of the TCN model. With the number of input channels constant, increasing the number of input layers can increase the number of parameters and thus improve the accuracy. However, the relationship between the number of parameters and the accuracy rate is not linear. When the number of parameters increases to a certain level, the accuracy rate is no longer improved, or the improvement is quite limited.

Therefore, two points are taken as the best model for TCN at the stage where the accuracy of the two sets of models plateau. For the TCN model with 64 input channels, when the number of input layers is 5 and the kernel size is 3, it is the best result with 94.115% accuracy and 116k parameters. For the TCN model with 128 input channels, when the number of input layers is 5 and the kernel size is 3, it is the best result with 95.026% accuracy and having 454k parameters.

As mentioned in Chapter 2, previous work has trained the KWS task with different neural network models. Therefore here, the performance of the two best TCN models mentioned above are compared with the previous models; see Table4.4.

Through comparison, it can be found that the TCN model is not much different from the previous model in terms of accuracy and is basically at the same level. However, the number of parameters is not advantageous: TCN-64 is only lower than DS-CNN-M and ResNet15 and consistent with LSTM but much larger than GRU, while TCN-128 is even more severe, and the number of parameters in 454k is even close to 10 times that of GRU.

Of course, it is not surprising that the TCN model has many parameters. Due to the presence of residual blocks and causal convolutions, TCNs have a large sensory field along with a deep number of network layers. The deeper network layers inevitably increase the number of parameters in the model.

Model	Parameters	Accuracy
DS-CNN-M	189k	94.9%
ResNet15	240k	95.8%
GRU-64	46k	95.1%
LSTM	118k	88.8%
TCN-64	116k	94.1%
TCN-128	454k	95.0%
DeltaTCN-1	454k	93.6%
DeltaTCN-2	454k	91.5%
SSPTCN-1	116k	93.6%
SSPTCN-2	116k	90.2%

Table 4.4: List of KWS performance under different models

4.3.2. DeltaTCN

To make the trade-off between lower network parameters and higher accuracy, the combination of 64 channels and 5 layers has been chosen to perform DeltaTCN retraining. The result is listed in Table 4.5.

Threshold	Accuracy	Sparsity
0	94.8479%	3.36%
0.1	93.6043%	62.08%
0.2	91.5390%	81.89%
0.3	89.8068%	87.56%
0.4	87.4306%	91.88%
0.5	86.1870%	96.56%

Table 4.5: Results of DeltaTCN network

As a reference, the result of DeltaTCN, when the threshold is 0, is used as the standard value. At this point, the accuracy is 94.8479%, and the initial sparsity is 3.36%. The accuracy of the TCN with the

same network structure before retraining is 94.115%, which is almost the same as the DeltaTCN with a threshold value of 0, which is consistent with the previous analysis.

As the threshold increases, the accuracy of DeltaTCN decreases while the sparsity increases. When the threshold is taken to 0.5, the accuracy decreases to 86.1870%, and the sparsity rate increases to 96.56%. At this point, the sparse rate is close to saturation.

According to table 4.5, the relationship between threshold and accuracy is shown in Fig. 4.7. The relationship of threshold and sparsity is shown in Fig. 4.8.

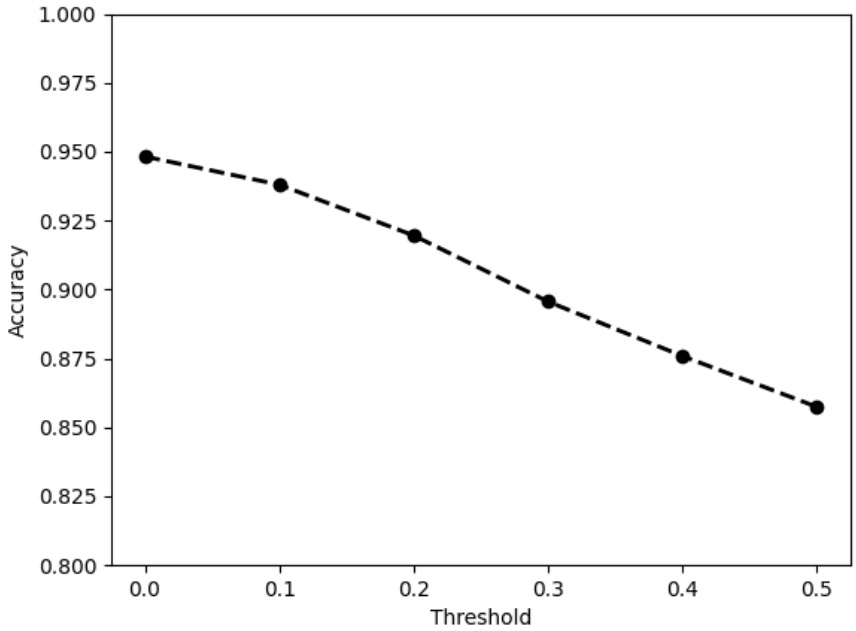


Figure 4.7: Threshold vs Accuracy(DeltaTCN model of 64 input channels, 5 input layers)

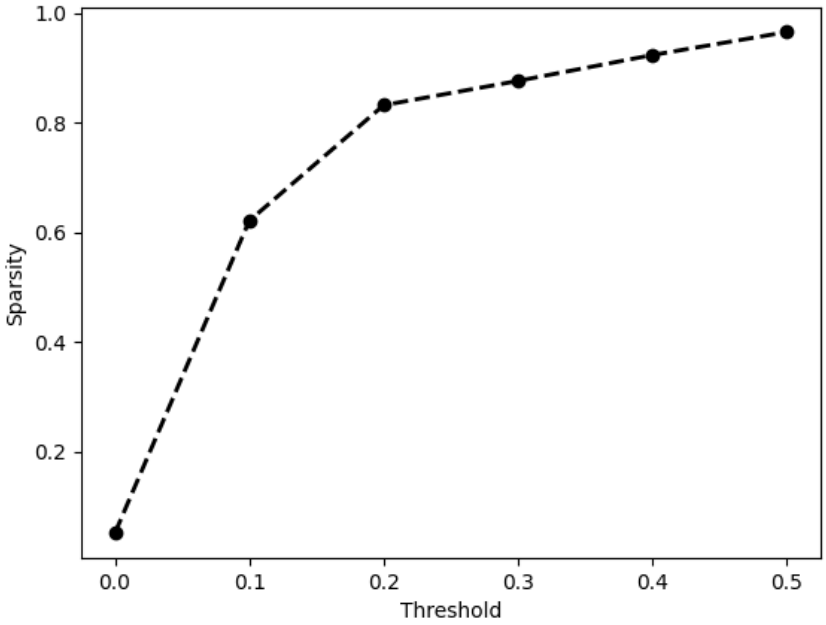


Figure 4.8: Threshold vs Sparsity(DeltaTCN model of 64 input channels, 5 input layers)

Similarly, the training results of DeltaTCN are compared with the previous model's performance. After constructing the Delta sparse layer, the computational effort of the model is significantly reduced, although the number of parameters of the DeltaTCN model is still large. Moreover, DeltaTCN can achieve relatively high sparsity, achieving considerable accuracy while significantly reducing the amount of computation.

However, DeltaTCN has major drawbacks. Since the target of constructing Delta sparse layer is the input of the convolutional layer, each step of convolutional computation needs to wait for the input data sparsification processing to be finished first, which leads to the training time of DeltaTCN is 8-10 times the ordinary TCN network, which greatly reduces the real-time of the KWS processing, and also brings a new growth of the power consumption of the device.

4.3.3. SSPTCN

A total of six sets of tests were done on the SSPTCN network; the results are shown in Fig. 4.9. According to the picture, regardless of which parameter setting the SSPTCN network is under, the accuracy decreases slowly at first as the sparsity increases and decreases significantly after a certain sparsity bottleneck is reached. This sparsity bottleneck is usually in the range of 50-60%. For the number of computations, as sparsity increases, the amount of network computation decreases dramatically.

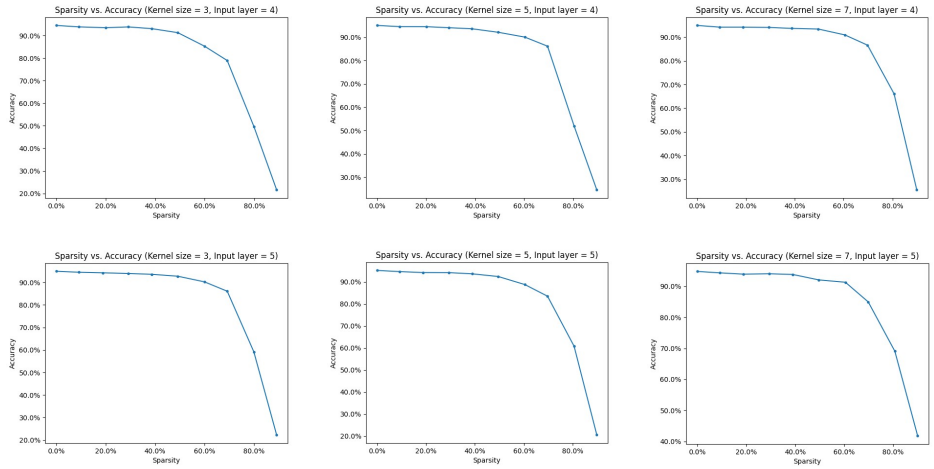


Figure 4.9: Sparsity vs. Accuracy of all sets of SSPTCNs, kernel size is 3,5,7, input channel number is 4,5, input channel is 64.

By comparing the accuracy and the number of network parameters, the SSPTCN network with 5 input layers and 3 kernel sizes was selected as the best result for analysis. The SSPTCN network with this parameter setting has the highest accuracy and smaller network parameters. The specific experimental results are shown in the Table4.6.

Accuracy	Sparsity	Parameter	FLOPs
94.937%	0.00%	116k	1.082G
94.499%	9.23%	105k	982M
94.254%	18.89%	94k	877M
93.947%	29.24%	82k	765M
93.599%	38.66%	71k	663M
92.740%	49.25%	59k	549M
90.245%	60.02%	46k	432M
86.074%	69.26%	36k	332M
59.162%	80.03%	23k	216M
22.209%	89.27%	12k	116M

Table 4.6: Best result for SSPTCN model(Input channel = 64, Input layer = 5, Kernel size = 3)

It can be seen that when the sparsity reaches 60%, the accuracy of the SSPTCN model decreases by 5% from 94.937% to 90.245%. This point is seen as the lowest point where the accuracy of the SSPTCN model is acceptable. And when the sparsity is only 38.66%, the accuracy is 93.599%, which is only 1.4% decrease compared to the network without sparsity. This point can be regarded as the equilibrium point of SSPTCN.

The performance of the SSPTCN model is compared with the previous model. It can be found that the performance of the SSPTCN model is almost the same as DeltaTCN in terms of accuracy. However, compared to DeltaTCN, SSPTCN outperforms DeltaTCN regarding the number of parameters and the amount of computation. This is because SSPTCN employs smaller input channels and a smaller number of input layers, which reduces the depth of the network compared to DeltaTCN.

What makes SSPTCN even better is the training time, which is almost the same as the regular TCN model. This is because the convolutional layers' weight matrix realizes how SSPTCN constructs the sparse layers. Compared to DeltaTCN, which constructs sparsity for the input data, SSPTCN does not have to wait for the sparsity matrix to be constructed to train but only needs to update the weight matrix as a whole after each cycle is completed. This not only saves a lot of training time and reduces the amount of computation but also does not affect the response speed of the model. This matches the realistic needs of the KWS task.

5 Conclusion

In this work, we first reproduce the performance of a vanilla TCN on the KWS task. By scanning different parameters, we found the optimal parameter settings for the vanilla TCN model and used them as the baseline values for this project. Also, after comparing the training results with previous neural network models (mainly RNNs) on the Google Speech Command V2 database, the standard TCN network used as a benchmark value matches the previous work regarding accuracy. Unfortunately, the standard TCN network is not superior regarding training parameters and computational effort. This is due to several reasons. First, the TCN model has many network layers, and to achieve a high accuracy rate, it adopts a 4 or 5 layer design, where each layer contains a temporal block. Each temporal block has the following components: convolutional layers and others, which introduce many additional parameters. Second, the TCN model uses residual connections, which also introduces additional parameters. Finally, dilated convolution adds flexibility but also additional parameters.

To reduce the training parameters and computation cost, we introduce temporal sparsity and weight sparsity in TCNs and construct DeltaTCN and SSPTCN based on them. For DeltaTCN, introducing temporal sparsity can create huge sparsity, thus saving a lot of computational resources. KWS tasks trained with DeltaTCN models cannot reduce the computation to be comparable to RNN-like models, but they can reduce the computation to be below ResNet, which also uses residual structure, with only about 1% accuracy loss. And things will be much better for SSPTCN. Because of the use of weight sparsity, SSPTCN can reduce not only the amount of computation but also the training parameters. With structured sparsity, SSPTCN is much smaller than RNN, LSTM, CNN, and ResNet models in training parameters and suffers only about 1% accuracy loss. Regarding computation, SSPTCN is an order of magnitude lower than the standard TCN model and at least 50% lower than ResNet with a similar residual structure.

For a KWS system, accuracy, response speed, and power consumption are the three most important metrics that users care about. This work verifies that the TCN model can fulfill the KWS training task well because TCN can provide the same accuracy as the most commonly used RNN model. Also, this work demonstrates that introducing different sparsification strategies to the TCN model can improve its performance, achieving significant optimization of training parameters and computation with little accuracy loss. This can be effectively realized for KWS devices regarding speed and power consumption.

5.1. Future Work

In this paper, two different sparsification approaches are proposed and applied to the TCN model respectively. In future work, one can try to combine the two and use them to construct the sparsification on both the input and weight matrices. Theoretically, this can further increase the sparsity of the model.

In this paper, the standard TCN model is reproduced, and only customized parameters, such as convolution kernel, input channel, etc., are adjusted, and the structure of the model is not changed. Therefore, the TCN model is not fully adapted to the training data. In future work, the structure of each

model layer can be customized to reduce the redundant parameters in each layer.

This project aims to reduce the computational and memory cost of TCN for the KWS task without losing much accuracy. This paper verifies the feasibility of the proposed methods at the software level, and future work can deploy these sparse TCNs to hardware to verify their effectiveness on hardware.

References

[1] Jan Robin Rohlicek et al. “Continuous hidden Markov modeling for speaker-independent word spotting”. In: *International Conference on Acoustics, Speech, and Signal Processing,* IEEE. 1989, pp. 627–630.

[2] Susanta Sarangi, Md Sahidullah, and Goutam Saha. “Optimization of data-driven filterbank for automatic speaker verification”. In: *Digital Signal Processing* 104 (Sept. 2020), p. 102795. ISSN: 1051-2004. DOI: 10.1016/j.dsp.2020.102795. URL: http://dx.doi.org/10.1016/j.dsp.2020.102795.

[3] Douglas A Reynolds and Richard C Rose. “Robust text-independent speaker identification using Gaussian mixture speaker models”. In: *IEEE transactions on speech and audio processing* 3.1 (1995), pp. 72–83.

[4] Marc Schröder. “Emotional speech synthesis: A review”. In: *Seventh European Conference on Speech Communication and Technology*. 2001.

[5] Yariv Ephraim and Harry L Van Trees. “A signal subspace approach for speech enhancement”. In: *IEEE Transactions on speech and audio processing* 3.4 (1995), pp. 251–266.

[6] Joseph P Campbell. “Speaker recognition: A tutorial”. In: *Proceedings of the IEEE* 85.9 (1997), pp. 1437–1462.

[7] Guoguo Chen, Carolina Parada, and Georg Heigold. “Small-footprint keyword spotting using deep neural networks”. In: *2014 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2014, pp. 4087–4091.

[8] George Tucker et al. “Model compression applied to small-footprint keyword spotting”. In: (2016).

[9] Vikas Sindhwani, Tara Sainath, and Sanjiv Kumar. “Structured transforms for small-footprint deep learning”. In: *Advances in Neural Information Processing Systems* 28 (2015).

[10] Sercan O Arik et al. “Convolutional recurrent neural networks for small-footprint keyword spotting”. In: *arXiv preprint arXiv:1703.05390* (2017).

[11] Yann LeCun, Yoshua Bengio, et al. “Convolutional networks for images, speech, and time series”. In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[12] Ossama Abdel-Hamid et al. “Applying convolutional neural networks concepts to hybrid NN-HMM model for speech recognition”. In: *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*. IEEE. 2012, pp. 4277–4280.

[13] László Tóth. “Combining time-and frequency-domain convolution in convolutional neural network-based phone recognition”. In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2014, pp. 190–194.

[14] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. *Learning internal representations by error propagation*. 1985.

[15] Niall O’Mahony et al. “Deep learning vs. traditional computer vision”. In: *Advances in Computer Vision: Proceedings of the 2019 Computer Vision Conference (CVC), Volume 1 1*. Springer. 2020, pp. 128–144.

[16] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and policy considerations for modern deep learning research”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 09. 2020, pp. 13693–13696.

[17] Michael Kohler and Adam Krzyzak. *Over-parametrized deep neural networks do not generalize well*. 2020. arXiv: 1912.03925 [math.ST].

[18] Chunyang Wu et al. “Improving interpretability and regularization in deep learning”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 26.2 (2017), pp. 256–265.

[19] Albert Zeyer et al. “A comprehensive study of deep bidirectional LSTM RNNs for acoustic modeling in speech recognition”. In: *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2017, pp. 2462–2466.

[20] Xiaodong Cui, Vaibhava Goel, and Brian Kingsbury. “Data augmentation for deep neural network acoustic modeling”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23.9 (2015), pp. 1469–1477.

[21] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[22] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[23] Emad A Ibrahim et al. “Keyword spotting using time-domain features in a temporal convolutional network”. In: *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE. 2019, pp. 313–319.

[24] Chiyuan Zhang et al. *Understanding deep learning requires rethinking generalization*. 2017. arXiv: 1611.03530 [cs.LG].

[25] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, 2011, pp. 315–323. URL: https://proceedings.mlr.press/v15/glorot11a.html.

[26] Christian Szegedy et al. “Going deeper with convolutions”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[27] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[28] Chuan Li. *OpenAI’s GPT-3 Language model: A technical overview*. Aug. 2023. URL: https://lambdalabs.com/blog/demystifying-gpt-3.

[29] Menglong Xu and Xiao-Lei Zhang. “Depthwise separable convolutional resnet with squeeze-and-excitation blocks for small-footprint keyword spotting”. In: *arXiv preprint arXiv:2004.12200* (2020).

[30] Trevor Gale, Erich Elsen, and Sara Hooker. “The state of sparsity in deep neural networks”. In: *arXiv preprint arXiv:1902.09574* (2019).

[31] Wei Wen et al. “Learning structured sparsity in deep neural networks”. In: *Advances in neural information processing systems* 29 (2016).

[32] Tzu-Hsien Yang et al. “Sparse reram engine: Joint exploration of activation and weight sparsity in compressed neural networks”. In: *Proceedings of the 46th International Symposium on Computer Architecture*. 2019, pp. 236–249.

[33] Sharan Narang et al. “Exploring sparsity in recurrent neural networks”. In: *arXiv preprint arXiv:1704.05119* (2017).

[34] Mark Kurtz et al. “Inducing and exploiting activation sparsity for fast inference on deep neural networks”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5533–5543.

[35] Bodo Rueckauer and Shih-Chii Liu. “Conversion of analog to spiking neural networks using sparse temporal coding”. In: *2018 IEEE international symposium on circuits and systems (ISCAS)*. IEEE. 2018, pp. 1–5.

[36] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. “An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling”. In: *CoRR* abs/1803.01271 (2018). arXiv: 1803.01271. URL: http://arxiv.org/abs/1803.01271.

[37] Gaurav Kumar and Pradeep Kumar Bhatia. “A detailed review of feature extraction in image processing systems”. In: *2014 Fourth international conference on advanced computing & communication technologies*. IEEE. 2014, pp. 5–12.

[38] Yun-Peng Wu, Jia-Min Mao, and Wei-Feng Li. “Robust speech recognition by selecting mel-filter banks”. In: *2nd Annual International Conference on Electronics, Electrical Engineering and Information Science (EEEIS 2016)*. Atlantis Press. 2016, pp. 407–416.

[39] Beth Logan et al. “Mel frequency cepstral coefficients for music modeling.” In: *Ismir*. Vol. 270. 1. Plymouth, MA. 2000, p. 11.

[40] Sean R Eddy. “Hidden markov models”. In: *Current opinion in structural biology* 6.3 (1996), pp. 361–365.

[41] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. “An application of recurrent neural networks to discriminative keyword spotting”. In: *International conference on artificial neural networks*. Springer. 2007, pp. 220–229.

[42] Tara Sainath and Carolina Parada. “Convolutional neural networks for small-footprint keyword spotting”. In: (2015).

[43] J.R. Rohlicek et al. “Continuous hidden Markov modeling for speaker-independent word spotting”. In: *International Conference on Acoustics, Speech, and Signal Processing,* 1989, 627–630 vol.1. DOI: 10.1109/ICASSP.1989.266505.

[44] R.C. Rose and D.B. Paul. “A hidden Markov model based keyword recognition system”. In: *International Conference on Acoustics, Speech, and Signal Processing*. 1990, 129–132 vol.1. DOI: 10.1109/ICASSP.1990.115555.

[45] J.G. Wilpon, L.G. Miller, and P. Modi. “Improvements and applications for key word recognition using hidden Markov modeling techniques”. In: 1991. DOI: 10.1109/icassp.1991.150338.

[46] Diego Vidaurre et al. “Discovering dynamic brain networks from big data in rest and task”. In: *NeuroImage* 180 (2018), pp. 646–656.

[47] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[48] A. Waibel et al. “Phoneme recognition using time-delay neural networks”. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.3 (1989), pp. 328–339. DOI: 10.1109/29.21701.

[49] Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. “A time delay neural network architecture for efficient modeling of long temporal contexts”. In: *Proc. Interspeech 2015*. 2015, pp. 3214–3218. DOI: 10.21437/Interspeech.2015-647.

[50] David Snyder, Daniel Garcia-Romero, and Daniel Povey. “Time delay deep neural network-based universal background models for speaker recognition”. In: *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. 2015, pp. 92–97. DOI: 10.1109/ASRU.2015.7404779.

[51] Jaeger S. et al. “Online handwriting recognition: the NPen++ recognizer”. In: *International Journal on Document Analysis and Recognition* 3.3 (Mar. 2001), pp. 169–180. ISSN: 1433-2833. DOI: 10.1007/pl00013559. URL: https://cir.nii.ac.jp/crid/1362825895158279168.

[52] C. Bregler et al. “Improving connected letter recognition by lipreading”. In: *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*. Vol. 1. 1993, 557–560 vol.1. DOI: 10.1109/ICASSP.1993.319179.

[53] Brecht Desplanques, Jenthe Thienpondt, and Kris Demuynck. “ECAPA-TDNN: Emphasized Channel Attention, Propagation and Aggregation in TDNN Based Speaker Verification”. In: *Interspeech 2020*. ISCA, Oct. 2020. DOI: 10.21437/interspeech.2020-2650. URL: https://doi.org/10.21437%2Finterspeech.2020-2650.

[54] Masoumeh Kalantari Khandani and Wasfy B. Mikhael. “Effect of Sparse Representation of Time Series Data on Learning Rate of Time-Delay Neural Networks”. In: *Circuits Syst. Signal Process.* 40.6 (June 2021), pp. 3007–3032. ISSN: 0278-081X. DOI: 10.1007/s00034-020-01610-8. URL: https://doi.org/10.1007/s00034-020-01610-8.

[55] Krzysztof J Cios. “Deep neural networks—A brief history”. In: *Advances in Data Analysis with Computational Intelligence Methods: Dedicated to Professor Jacek Żurada* (2018), pp. 183–200.

[56] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural networks* 61 (2015), pp. 85–117.

[57] Alex Sherstinsky. “Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network”. In: *Physica D: Nonlinear Phenomena* 404 (2020), p. 132306.

[58] Kunihiko Fukushima. “Neocognitron: A hierarchical neural network capable of visual pattern recognition”. In: *Neural Networks* 1.2 (1988), pp. 119–130. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(88)90014-7. URL: https://www.sciencedirect.com/science/article/pii/0893608088900147.

[59] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[60] G. E. Hinton and R. R. Salakhutdinov. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), pp. 504–507. DOI: 10.1126/science.1127647. eprint: https://www.science.org/doi/pdf/10.1126/science.1127647. URL: https://www.science.org/doi/abs/10.1126/science.1127647.

[61] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[62] Destiny Ogaga and Abiodun Olalere. “Evaluation and Comparison of SVM, Deep Learning, and Naı□ve Bayes Performances for Natural Language Processing Text Classification Task”. In: (2023).

[63] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[65] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Binaryconnect: Training deep neural networks with binary weights during propagations”. In: *Advances in neural information processing systems* 28 (2015).

[66] Song Han, Huizi Mao, and William J Dally. “DEEP COMPRESSION: COMPRESSING DEEP NEURAL NETWORKS WITH PRUNING, TRAINED QUANTIZATION AND HUFFMAN CODING”. In: *arXiv preprint arXiv:1510.00149* (2015).

[67] Mohammad Rastegari et al. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[68] Dongqing Zhang et al. “Lq-nets: Learned quantization for highly accurate and compact deep neural networks”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 365–382.

[69] Yu Bai, Yu-Xiang Wang, and Edo Liberty. “ProxQuant: Quantized Neural Networks via Proximal Operators”. In: *International Conference on Learning Representations*. 2018.

[70] Shilin Zhu, Xin Dong, and Hao Su. “Binary ensemble neural network: More bits per network or more networks per bit?” In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 4923–4932.

[71] Yann A. LeCun et al. “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[72] Song Han et al. “Learning Both Weights and Connections for Efficient Neural Networks”. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’15. Montreal, Canada: MIT Press, 2015, pp. 1135–1143.

[73] Yang He et al. “Filter pruning via geometric median for deep convolutional neural networks acceleration”. In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 4340–4349.

[74] Huizi Mao et al. “Exploring the Granularity of Sparsity in Convolutional Neural Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2017, pp. 1927–1934. DOI: 10.1109/CVPRW.2017.241.

[75] Wei Niu et al. “PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-based Weight Pruning”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Mar. 2020. DOI: 10.1145/3373376.3378534. URL: https://doi.org/10.1145%2F3373376.3378534.

[76] Yihui He et al. “Amc: Automl for model compression and acceleration on mobile devices”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 784–800.

[77] Hao Li et al. *Pruning Filters for Efficient ConvNets*. 2017. arXiv: 1608.08710 [cs.CV].

[78] Huizi Mao et al. *Exploring the Regularity of Sparse Structure in Convolutional Neural Networks*. 2017. arXiv: 1705.08922 [cs.LG].

[79] Jiaqi Zhang et al. “Eager Pruning: Algorithm and Architecture Support for Fast Training of Deep Neural Networks”. In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 2019, pp. 292–303.

[80] Dingqing Yang et al. “Procrustes: a dataflow and accelerator for sparse deep neural network training”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 711–724.

[81] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel pruning for accelerating very deep neural networks”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 1389–1397.

[82] Sanghoon Kang et al. “An Overview of Sparsity Exploitation in CNNs for On-Device Intelligence With Software-Hardware Cross-Layer Optimizations”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 11.4 (2021), pp. 634–648. DOI: 10.1109/JETCAS.2021.3120417.

[83] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[84] Evangelos Stromatias et al. “Robustness of spiking deep belief networks to noise and reduced bit precision of neuro-inspired hardware platforms”. In: *Frontiers in neuroscience* 9 (2015), p. 222.

[85] Pete Warden. “Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition”. In: *CoRR* abs/1804.03209 (2018). arXiv: 1804.03209. URL: http://arxiv.org/abs/1804.03209.

[86] Iván López-Espejo et al. “Deep spoken keyword spotting: An overview”. In: *IEEE Access* 10 (2021), pp. 4169–4199.