Deep Learning for Electrocardiogram (ECG) Identification

by

Ziyi Tian
B.Eng, , Beijing University of Technology , 2018

A Project Report Submitted in Partial fulfillment of the
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Ziyi Tian, 2020
University of Victoria

All rights reserved. This project may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Supervisory Committee

Deep Learning for ECGs Recognition

by

Ziyi Tian
B.Eng, , Beijing University of Technology , 2018

Supervisory Committee

Dr. Wu-Sheng Lu, Department of Electrical and Computer Engineering, University of Victoria
Supervisor

Dr. XiaoDai Dong, Department of Electrical and Computer Engineering, University of Victoria
Member

Abstract

In this report, we describe three architectures based on deep learning for electrocardiogram (ECG) identification. Specially, a class of neural network-convolutional network-is used both to extract features from ECG signals and do classification. We provide necessary details in this report for implementation of these networks by using Keras and applying them on a publicly available dataset-MIT-BIH dataset, which contains 48 half-hour excerpts of two-channel ambulatory ECG recordings, obtained from 47 subjects studied by the BIH Arrhythmia Laboratory between 1975 and 1979. This report also shows the accuracy and computing time of our architectures on MIT-BIT datasets and comparison with other methods. The effect of several techniques specifically designed for CNNs are also discussed..

Keywords: ECG, Deep Learning, Convolutional Neural Network

Table of Contents

Abstract ..... **ii**

ACKNOWLEDGEMENTS ..... **v**

List of Tables ..... **vi**

List of Figures ..... **vii**

Abbreviations ..... **ix**

Chapter 1 INTRODUCTION **1**

1.1 What Is Electrocardiogram? 1

1.2 Basic Architectures of Neural Networks 3

1.3 Machine Learning Technologies for ECG-Based Prediction 7

Chapter 2 ..... **8**

Convolutional Neural Networks for Classification of ECG Signals **8**

2.1 A Brief History of Deep Learning 8

2.2 Basic Architectures of CNNs 9

2.2.1 Convolutional layers 10

2.2.2. Activation layers 12

2.2.3 Pooling layers 13

2.2.4 Fully connected layers and output layers 15

2.3 Techniques Specifically Designed for CNNs 16

2.3.1 Batch normalization 16

2.3.2 Shortcut 17

2.3.3 Dropout 18

2.4 Our Models 20

2.4.1 Architecture 20

2.4.2 Implementation 22

Chapter 3 Performance Evaluation ... 23

3.1 The Datasets ... 23

3.1.1 The MIT-BIH database ... 23

3.1.2 Data Preparation ... 24

3.2 Experimental Results ... 25

3.2.1 Three CNNs based models ... 25

3.2.2 Feature maps ... 30

3.2.3 Architectures without batch normalization ... 32

3.2.4 Networks without shortcut ... 36

3.2.5 Networks with different dropping rate ... 41

3.2.6 Comparisons with other techniques ... 42

Chapter 4 Conclusions ... 44

Appendix Python Codes ... 45

A. Code for Configuration ... 45

B. Code for 9-layer Network ... 45

C. Code for 19-layer Network ... 48

C. Code for 35-layer network ... 50

D. Code for Data Preprocessing ... 52

E. Code for Training and Testing ... 53

Reference ... 55

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Lu, who guided me throughout this project. Without his invaluable suggestions, patience and support, I could not have completed this project.

I would also like to thank my family and friends who supported and encouraged me during my study.

List of Tables

Table 3.1. Datasets Description 23

Table 3.2. Classification accuracy of three CNN-based models 26

Table 3.3. Training and testing time of three CNN-based models 27

Table 3.4. Classification accuracy versus splitting ratio 29

Table 3.5. Classification accuracy of three CNNs without batch normalization 33

Table 3.6. Training and testing time of three CNNs without batch normalization. 34

Table 3.7. Classification accuracy of three models without shortcut. 37

Table 3.8. Training and testing time of three models without batch normalization 38

Table 3.9. Average training and testing accuracy of CNNs with different dropping rates. 41

Table 3.10. Performance of several classifiers using the same database. 42

List of Figures

Figure 1.1. Placement of electrocardiogram (ECG) electrodes. 2

Figure 1.2. ECG of a heart in normal sinus rhythm. 2

Figure 1.3. Biological neural network. 3

Figure 1.4. Simplest architecture of neural network. 3

Figure 1.5. Perception without bias. 4

Figure 1.6. Perception with bias. 5

Figure 1.7. A single-layer neural network. 5

Figure 1.8. A neural network with two hidden layers and a single output. 6

Figure 2.1. Local receptive field 1. 10

Figure 2.2. Local receptive field 2. 11

Figure 2.3. A convolutional hidden layer with three feature maps. 12

Figure 2.4. Activation functions. 13

Figure 2.5. Max pooling. 14

Figure 2.6. Average pooling. 14

Figure 2.7. Size changes when a convolutional layer is combined with a pooling layer. 15

Figure 2.8. A CNN with a fully connected layer. 15

Figure 2.9. Degradation of deeper networks. 17

Figure 2.10. Residual learning: a building block. 18

Figure 2.11. Dropout neural network. 18

Figure 2.12. Units in dropout neural networks. 19

Figure 2.13. Comparison of the basic operations of a standard a dropout network with dropout 20

Figure 2.14. Architectural illustration of the three CNNs. 21

Figure 3.1. Confusion matrices averaged over 20 datasets. 29

Figure 3.2. The nth feature maps in the 9-layer network. 30

Figure 3.3. The nth feature maps in the 19-layer network. 31

Figure 3.4. The nth feature maps in the 35-layer network. 32

Figure 3.5. Architectural illustration of the three CNNs without batch normalization. 36

Figure 3.6. Architectural illustration of the three CNNs without shortcut. 40

Abbreviations

CNN	Convolutional Neural Network
DBLSTM	Deep Bidirectional Long Short-Term Memory
DSP	Digital Signal Processing
ECG	Electrocardiogram
GPU	Graphics Processor Unit
LSTM	Long Short-Term Memory
NN	Neural Network
PCAnet	PCA network
ReLU	Rectified Linear Unit
SVM	Support Vector Machine
tanh	Hyperbolic Tangent
1D-CNN	One-Dimensional Convolutional Neural Network

Chapter 1 Introduction

Electrocardiograms (ECGs) have been widely used for reflecting heart conditions, and inspection of their patterns has found helpful in facilitating diagnosis of heart diseases. Unlike other biometric modalities such as fingerprint, face, iris, and signature, ECGs are continuous time series whose analysis can be of use if ECG recordings over a sufficient time duration are available. In effect, the utilization of computer-based methods for automatic processing ECG signals can benefit cardiologists in their diagnosis. Over the past decades, many methods have been proposed for arrhythmia detection [1-8]. Several of these approaches are based on digital signal processing (DSP) techniques and classification algorithms such as support vector machines (SVMs). Recently, machine learning methods based on deep neural network architectures are emerging as powerful means in a variety of applications including computer vision and natural language processing. One of the most successful deep learning architectures is the convolutional neural networks (CNNs). In this report, we explore the potential of CNNs in detection and classification of arrhythmias using ECG signals. To start, this chapter provides a background of ECGs and neural networks.

1.1 What Is Electrocardiogram?

Electrocardiogram (ECG) is a record of electrical activities of the heart. It is usually given in the form of a graph of voltage values versus time, and is produced by using electrodes which are placed on the skin to detect electrical changes caused by depolarization and repolarization of cardiac muscle during each cardiac circle [9]. For example, a 12-lead ECG is generated by ten electrodes that are placed on the surface of patients' limbs and chest, as shown in Figure 1.1. Electrical potential of the heart is then captured from the leads and recorded over a number of cardiac circles [9].

There are mainly three major deflections that record the sequence of electrical propagation, namely the P wave, QRS complex, and T wave. The P wave represents the depolarization of the atria. Following the P wave, the tracing returns to its baseline. The QRS complex represents the depolarization of the ventricles. After the QRS complex, the tracing returns to baseline once again, and after a brief delay, repolarization of the ventricular cells is signaled by the T wave. Occasionally, an additional small deflection follows the T wave (called U wave), which is believed to represent a late phase of ventricular repolarization [9].

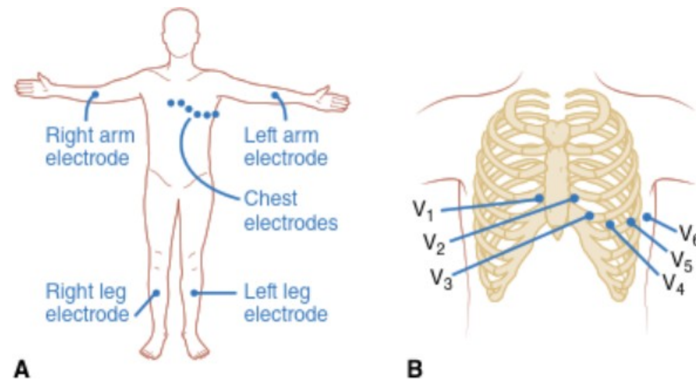


Figure1.1. Placement of electrocardiogram (ECG) electrodes: (A) Standard positions, (B) Close-up view of chest electrodes placement [9].

For patients with heart diseases, a variety of cardiac abnormalities may occur and these abnormalities usually lead to changes in ECG patterns. The abnormalities may include cardiac rhythm disturbances (such as atrial fibrillation and ventricular tachycardia), inadequate coronary artery blood flow (such as myocardial ischemia and myocardial infarction), and electrolyte disturbances (such as hypokalemia and hyperkalemia) [9]. In this project, we will focus on using CNN-based models for the detection of different types of abnormalities that are collected in the ECGs from a public database.

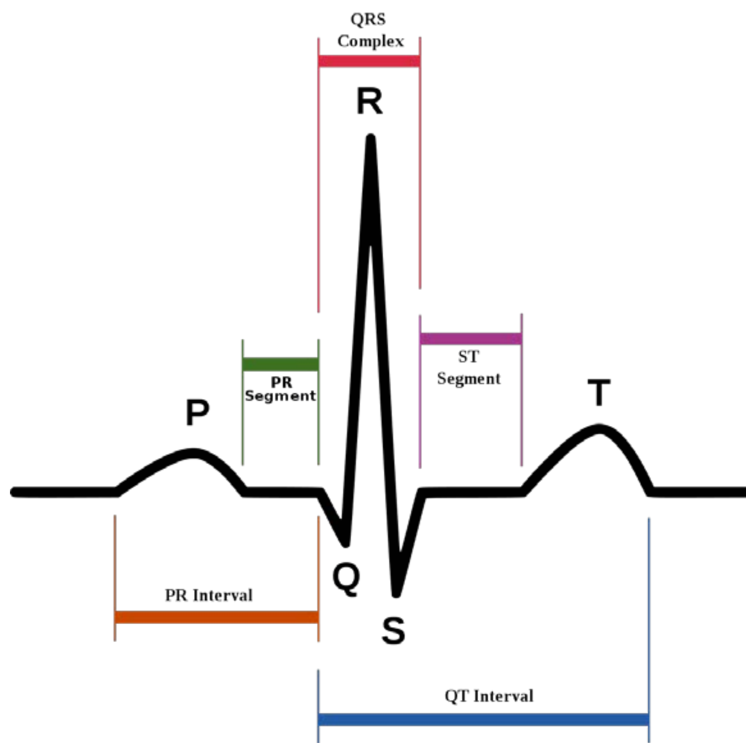


Figure 1.2. ECG of a heart in normal sinus rhythm [10].

1.2 Basic Architectures of Neural Networks

Neural networks, or more precisely artificial neural networks are networks for simulation of learning process in human brains. The human nervous system consists of neurons, which are connected to each other by means of axons, which are used for transmitting signals and dendrites, which receive signals from other neurons. Figure 1.3 illustrates connections between two neurons. The connection region between axons and dendrites is referred as synapses. The strengths of each synaptic connection may change depending on the external simulated. These changes are referred to as learning in human brains [11].

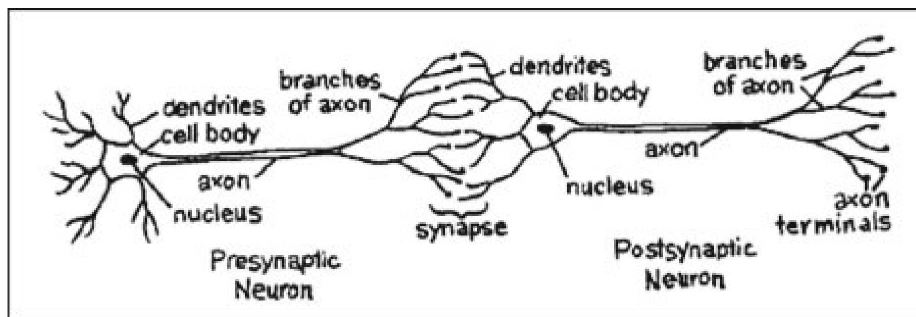


Figure 1.3. Biological neural network [11].

Artificial neural networks simulate this biological mechanism. For convenience, throughout this report, artificial neural networks will be referred to as “neural network”. A simplest architecture of neural network is illustrated in Figure 1.4 where a neuron combines inputs with *adjustable*

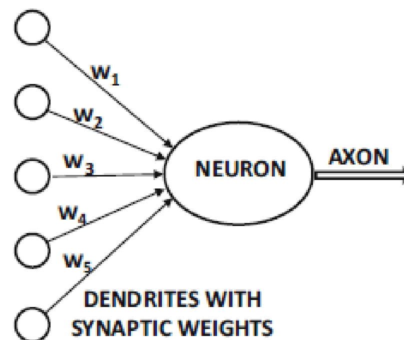


Figure 1.4. Simplest architecture of neural network [11].

weights $\{w_i\}$ and feeds the combination to an *activation* function to generate an output. Just like the strength changes of synaptic connections in response to external stimuli, values of these weights are modified during a *training* process using labeled training data. Afterwards the network is believed to mimic the function of a neuron in biological organisms [11].

The above neural network is often referred to as *perception*. Figure 1.5 shows a perception without bias. In this architecture, he input layer contains d inputs $\boldsymbol{x} = [x_1 \ldots x_d]$, which are transmitted to computation units with weights $\boldsymbol{w} = [w_1 \ldots w_d]$, and the linear function

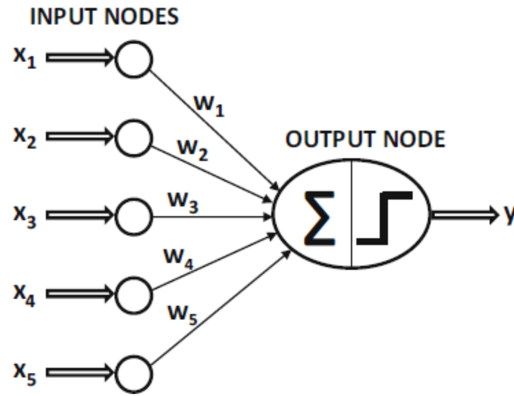


Figure 1.5. Perception without bias [11].

$$\boldsymbol{w} \cdot \boldsymbol{x} = \sum_{i=1}^{d} w_i x_i$$

is used to combine weighted inputs. Following this, the sign function

$$f(z) = \begin{cases} 1 & \text{for } z \geq 0 \\ -1 & \text{for } z < 0 \end{cases}$$

is applied as activation to yield an output y, namely,

$$y = \text{sign}\left(\sum_{i=1}^{d} w_i x_i \right)$$

We remark that the sign function is non differentiable at $z = 0$, several other types of activation functions will be discussed in Section 2.2.2.

In the case where the dataset is highly imbalanced, prediction performance of the above architecture may be degraded. For example, the correct mean value of an imbalanced binary dataset with class labels –1 or 1 is nonzero, and in this case the above network needs an invariant *bias b* to deal with the problem, see Figure 1.6.

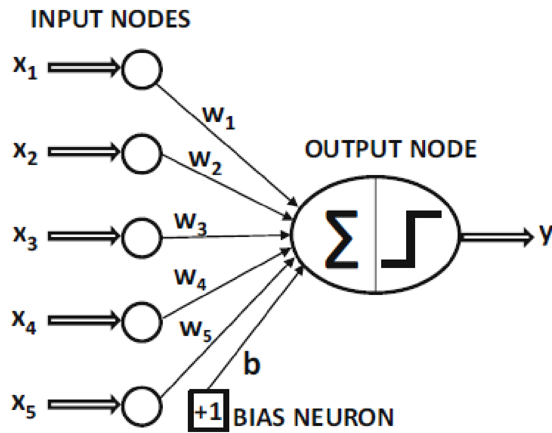


Figure 1.6. Perception with bias [11].

The input-output relationship of the network in Fig. 1.6 is given by

$$y = \text{sign}\left(\sum_{i=1}^{d} w_i x_i + b \right)$$

By combining a total of S perceptions, a single-layer neural network can be constructed, see Figure 1.7.

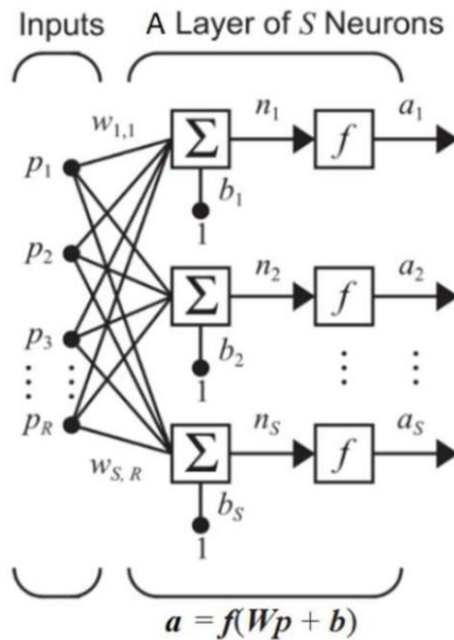


Figure 1.7. A single-layer neural network [20].

In this network, the ith neuron receives R inputs $\{p_j, j = 1, 2, \ldots, R\}$ that are combined with relevant weights to produce n_i which is then fed into an activation function f to generate output a_i. If we let

$$\boldsymbol{p} = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_R \end{bmatrix}, \boldsymbol{w}_i = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}, \boldsymbol{W} = \begin{bmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \\ \vdots \\ \boldsymbol{w}_S^T \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix}, \boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix}$$

the network structure can be characterized as

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_S \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{w}_1^T \boldsymbol{p} + b_1) \\ f(\boldsymbol{w}_2^T \boldsymbol{p} + b_2) \\ \vdots \\ f(\boldsymbol{w}_S^T \boldsymbol{p} + b_S) \end{bmatrix} = \boldsymbol{f}\left(\boldsymbol{W}\boldsymbol{p} + \boldsymbol{b}\right)$$

A neural network may have more than one computational layer. A multi-layer network with two hidden layers, each has its own biases and weight matrix, is depicted in Fig. 1.8.

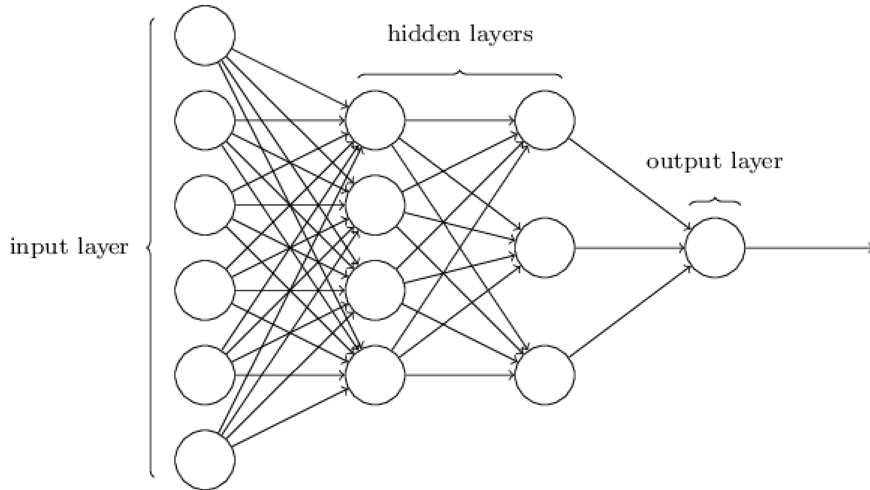


Figure 1.8. A neural network with two hidden layers and a single output [20].

For a general multilayer network with k hidden layers, there are $k + 1$ sets of parameters which are denoted by $\{\boldsymbol{W}_i, \boldsymbol{b}_i\}$ for $i = 1, 2, \ldots, k + 1$, and the input-output relationship of the network can be characterized by the recursive equations [11]

$$\begin{aligned} &\boldsymbol{a}_1 = \boldsymbol{f}_1(\boldsymbol{W}_1 \boldsymbol{p} + \boldsymbol{b}_1) && \text{input to hidden layer 1} \\ &\boldsymbol{a}_{i+1} = \boldsymbol{f}_{i+1}(\boldsymbol{W}_{i+1} \boldsymbol{a}_i + \boldsymbol{b}_{i+1}) \quad \text{for } i = 1, \ldots, k-1 && \text{hidden layer } i \text{ to hidden layer } i+1 \\ &\hat{\boldsymbol{y}} = \boldsymbol{f}_{k+1}(\boldsymbol{W}_{k+1} \boldsymbol{a}_k + \boldsymbol{b}_{k+1}) && \text{hidden layer } k \text{ to output layer} \end{aligned}$$

where $\boldsymbol{a}_i$ for $i = 1, 2, \ldots, k - 1$ donates the output of the ith hidden layer, $\boldsymbol{a}_k$ donates the output of the last hidden layer, $\boldsymbol{W}_{i+1}$ and $\boldsymbol{b}_i$ donates the weight and bias of the ith hidden layer, $\boldsymbol{p}$ donates the

input of this network and $\hat{y}$ donates the output of the network. As excepted, multilayer neural networks are able to provide improved performance relative to single-layer neural networks, especially when dealing with large-scale datasets [11].

1.3 Machine Learning Technologies for ECG-Based Prediction

Over the years many ECG-based methods have been proposed for arrhythmia detection [1-8]. In general, a method in this class consists of three main steps, namely prepossessing, feature extraction, and classification. These methods differ from each other in the way the feature extraction and classification are performed. In general, the feature extraction methods include, but not limited to, morphology [2], temporal information [3], high-order statistics [4], Hermite basis functions [5], hidden Markov modeling (HMM) [6] and convolutional neural networks (CNNs) [7]. Ref. [7] describes a method achieving an 99.3% accuracy on the PTB Diagnostic ECG Database, where CNN is used for feature extraction and support-vector machine (SVM) is used for classification.

Ref. [8] reports state-of-the-art results of a CNN-based deep learning architecture for ECG-based prediction of heart abnormalities, where its performance was compared with that of six cardiologists to demonstrate that the model outperforms in average the cardiologists' performance in both recall and precision. The work of [8] demonstrated that CNN-based deep learning holds promise for arrhythmia detection from ECG signals. Inspired by the work in [8], in this project we develop three models and evaluate their performance on the MIT-BIH data sets.

Chapter 2

Convolutional Neural Networks for Classification of ECG Signals

Convolutional neural network (CNN) is one of the first successful architectures of deep learning, especially for classification of images, video, texts, and speech. In this chapter, we focus on the application of CNNs to classification of ECG signals.

2.1 A Brief History of Deep Learning

Deep learning is a branch of machine learning methods based on multi-layer neural networks, where the algorithm development is highly motivated by the thinking process of humans, typically featuring the use of considerable number of hidden layers, where each layer is 3-dimensional with a spatial extent and a depth corresponding to the number of features [11]. The history of deep learning can be traced back to 1943 when Pitts and McCulloch proposed a model by mimicking neural networks of human brain [12]. In this model, a mathematical model called "threshold logic" imitates the thinking process in human brains. Unfortunately, afterwards the progress made in research of deep learning was rather slow, with only handful significant breaks during the next 30 years. In 1960, a continuous back propagation algorithm was derived by Kelley [13] and a simpler version of it based on the chain rule was proposed by Dreyfus [14]. Ivakhnenko and Lapa developed the fort model based on deep learning [15]. In their models, polynomial activation functions are employed and analyzed statistically, in each layer statistically optimal features are extracted and forwarded to the next layer.

From 1974 to 1980 and 1985 to 1990s, lack of funding limited the development of deep learning, and researchers experienced tough time known as the first and second AI winter. Fortunately, several researchers continued their research even without enough funding. In 1980, Fukushima proposed *neocognitron*, which is considered as an initial architecture of convolutional neural network [16]. The model was trained according to a reinforcement strategy which recurring activation in multiple layers. Moreover, the weights in certain connections are allowed to change in order to adjust important features. In 1997, the technique known as long-short-memory was developed by Hochreiter and Schmidhuber [17]. The next significant evolution in deep learning took place in late 1990s, when advanced computer hardware including graphics processing units (GPU) became available. Parallel to the advances in computer hardware, support vector machine (SVM) was proposed and shown to be an efficient machine leaning algorithm for pattern recognition.

In addition to the continuing growth in computational power, the recent years have seen an increase in data availability growth of data which also stimulated the development of deep learning. As the amount of data increases, neural networks have an advantage over conventional machine learning algorithms because they retain the flexibility to model more complex functions with the addition of neurons to the computational graph [11]. Currently there are about 2.5 quintillion bytes of data created each day. A prediction made by IDC says that worldwide data creation will grow to an enormous 163 zettabytes by 2025, which is ten times the amount of data produced in 2017 [18].

By 2010, with the significant increase of computer's speed made it is possible to train a convolutional neural network without the layer-by-layer pre-training. In 2011 and 2012, a CNN-based model known as AlexNet won the Large Scale Visual Recognition Challenge (ILSVRC) [19]. Since then CNNs have been widely used in computer vision. In 2019, the Res2Net achieves 94.37% accuracy on the ImageNet dataset [20], exceeding that of humans. Currently, deep learning continues to evolve and demonstrates its power in a variety of applications.

The earliest motivation of CNNs was obtained from Hubel and Wiesel's understanding of the workings of the cat's visual cortex, in which specific portions of the visual field seemed to excite particular neurons [21], [11]. Inspired by the discovery, *neocognitron* was proposed by Fukushima in 1980 [16], which is considered by many as the earliest CNN model. In 1989, the first modern framework of CNNs, known as LeNet-5, was proposed by LeCun et al [22]. CNNs have since evolved rapidly over the past decades, and have been the most successful of all types of neural networks. They are used widely for image recognition, object detection/localization, and text processing. Recently, application of CNN architectures has been extended to the classification of ECG signals [7], [8].

2.2 Basic Architectures of CNNs

The states in each layers of a CNN are arranged according to a spatial grid structure. The value from each output of the node is derived from a small local spatial region in the previous layer, and these spatial relationships are inherited from one layer to the next [11]. For classification of images, each layer in the CNN has three dimensions, which are *height*, *weight* and *depth*. For ECG signals, however, the input is a one-dimensional (1D), each layer of the CNN has a two-dimensional

architecture, namely height and depth. The height of a layer in CNN refers to the length of the input signals while the term "depth" refers to the number of channels in each layer.

There are three types of layers in a CNN, namely convolutional layers, pooling layers and activation layers. Associated with these layers there are additional signal paths and procedures for performance enhance, these include shortcut, batch normalization, and dropout. The architectural details of CNNs are described in the rest of the chapter.

2.2.1 Convolutional layers

The operation of *convolution* is the defining characteristic of CNNs. The mathematical essence of convolution is an inner produce of two vectors (or matrices, or multidimensional arrays) of same size, where one of the vectors involves a set of *weights*, which is usually referred to as *kernel*, and the other vector is a set of local samples from the input, called *local receptive field*. The inner product is always performed with the kernel fixed, while the local receptive field *shifts* in order to cover the entire input [21], see Figs. 2.1 and 2.2 for illustration for the case with one stride.

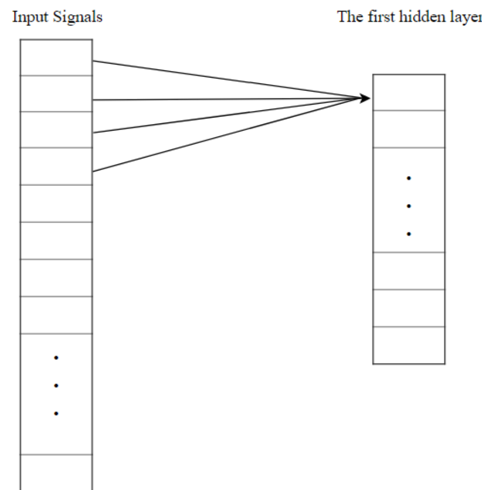


Figure 2.1. Local receptive field 1.

Usually the size of the kernel is small relative the length of the input in order to ensure that local features from the input are extracted. If we let the input be $\boldsymbol{x} = \{x_i, i = 0,1,...,n-1\}$ and the kernel be of length L and denoted by $\boldsymbol{w} = \{w_l, l = 0,1,..., L-1\}$, the 1D convolution $\boldsymbol{y} = \boldsymbol{w} \otimes \boldsymbol{x}$ is defined by

$$y_i = \sum_{l=0}^{L-1} w_l x_{i+l}$$

where index i for a full-scale one-stride convolution varies from 0 to $n - L$. For a full-scale S-stride convolution, the increment in index i is S, so i shall vary from 0 to floor $((n - L)/S)$ where floor(p) denotes the largest integer not greater than p.

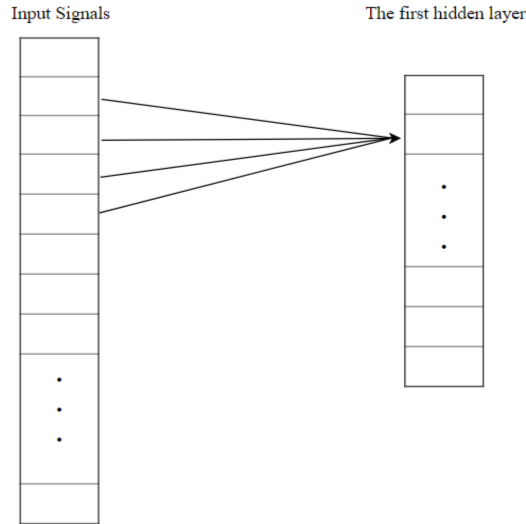


Figure 2.2. Local receptive field 2.

As a result, the one-stride convolution yields a vector with $L - 1$ less components compared to the input. For example, with $L = 9$ and an ECG signal of length 512 from the MIT-BIH dataset, the convolutional operation produces a vector of size 504 by 1.

The output of the convolution is then fed into an activation function f to yield

$$a_i = f\left(\sum_{l=0}^{L-1} w_l x_{i+l} + b\right)$$

For detailed discuss of activation, please refer to Section 2.3.2. Since the output of activation function contains local features of input signal, the vector $\{a_i\}$ is called a *feature map*. In order to extract features from input data consistently, it is important that all local convolution operations use the *same* weight vector $\boldsymbol{w}$ and bias b. This strategy refers to as *shared weights* and *shared bias*. Obviously, $\boldsymbol{w}$ and b must be chosen properly in order for the CNN to perform properly. This is done by training the $\{\boldsymbol{w}, b\}$ using labeled input data. At this point we stress that convolution with a pair of trained $\{\boldsymbol{w}, b\}$ only extracts *one* feature map, and for the CNNs for ECG classification problems to perform well it is necessary employ multiple kernels, $\{\boldsymbol{W}_i, b_i\}$ for $i = 1, 2, \ldots, J$ in

each given hidden layer of the CNN, see Fig. 2.3 for illustration where three feature maps are produced by three distinctly different convolutional kernels.

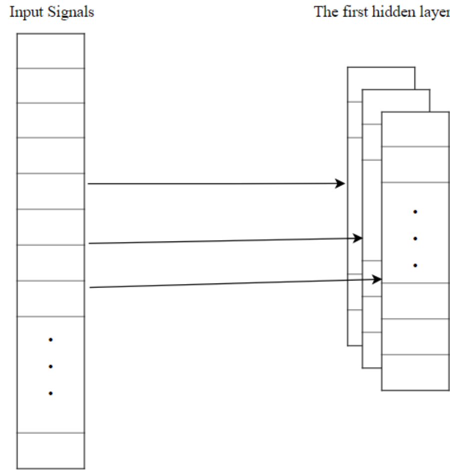


Figure 2.3. A convolutional hidden layer with three feature maps.

On comparing to fully connected neural networks (NNs), the number of parameters in CNNs is drastically reduced. For illustration of this complexity reduction, consider an input of size 512 by 1 that is fed into a hidden layer with $J = 18$ and $L = 9$ to generate 18 feature maps. In this case, a total $(9 + 1) \times 18 = 180$ parameters are involved in the layer, while a fully connected NN with 30 neurons uses $(512 + 1) \times 30 = 15390$ parameters, 84 times more than those involved in the CNN.

We now conclude this section with a remark on an issue concerning full-scale convolutions. By definition the size of a full-scale convolution is reduced by $L - 1$ compared to that of the input. The size reduction is not a desirable one in some implementations, but the problem can be fixed by padding zeros at both end of the input sequence before the convolution is performed. The number of zeros involved at each end in this action is referred to as the *padding size* [11].

2.2.2. Activation layers

In Section 1.2, the sign function was introduced as an activation function. In addition to this function, a variety of others types of activation functions are available such as sigmoid function, hyperbolic tangent (**`tanh`**) defined by

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

and *rectified linear unit* (ReLU) which yields **`max(0,x)`** in respond to input x. These and several

other popular activation functions are depicted in Figure 2.4 [21].

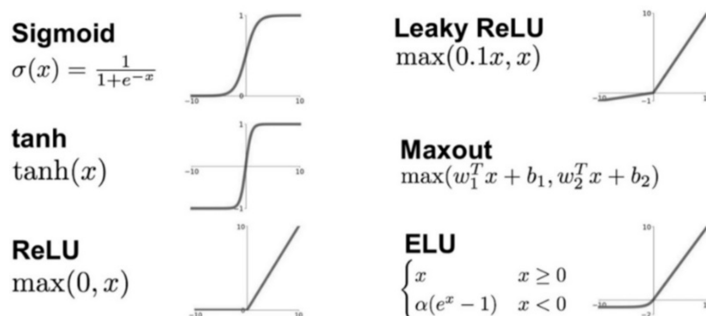


Figure 2.4. Activation functions [21].

In recent years, ReLUs have been employed more often than others in deep learning architectures. A ReLU function is applied to each node of a layer to generate threshold values of its inputs which are then passed on to the next layer which is typically another convolutional layer.

The use of ReLUs has been witnessed as an evolution in neural networks. By definition, a ReLU produces a zero value for a negative input while preserves a positive input without modification. On comparing it with other activation functions such as sigmoid and tanh, ReLUs has much reduced computational complexity, especially because few neurons would be activated simultaneously at any given time. It is generally believed that a CNN employing ReLUs can be as six times faster as a CNN employing sigmoid and tanh [21]. The work of [19] argues that the use of ReLUs has significant advantages over other activation functions both in terms of speed and accuracy.

2.2.3 Pooling layers

A convolutional layer in a CNN is always followed by a *pooling* layer, aiming to further reduce complexity and improve robustness of the network which refers to the network's reduced sensitivity to spatial variations of the input [21]. The operation of pooling first divides a feature map into sub-regions of a small size $P_q \times 1$ and selects *one* representative value from each region.

In this way, the action of pooling yields a corresponding feature map of much reduced size. It is important to mentioned that a sub-region overlaps the next one, and the degree of overlaps is measured by the amount of shift from one sub-region to the next and is referred to as *stride length*.

Average and max pooling are the two of the most common pooling operations. Figure 2.5 shows a simple example of max pooling where the maximum sample value within a given sub-region is

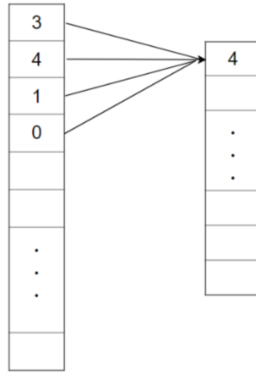


Figure 2.5. Max pooling.

selected. If a stride of 1 is used, the max pooling from the q-th convolutional layer yields a new layer of size $(L_q - P_q + 1) \times d_q$ where L_q donates the height of input signal, P_q donates pooling size and d_q donates the number of this new layer's channels. If the stride length $S_q > 1$, the size of new layer is equal to

$$\left(\frac{L_q - P_q}{S_q} + 1 \right) \times d_q$$

Average pooing is widely used in early work of CNNs such as LeNet-5, and was referred to as *subsampling*. The average pooling selects the average value over each sub-region, a simple example of average pooling is shown in Figure 2.6.

Figure 2.7 illustrates the size changes of a 1D signal that are induced by employing convolution and pooing operations, where three 9 × 1 kernels are used in a convolutional layer to generate three feature maps of size 504, which is followed by a max-pooling layer of size 2 × 1 and stride of 2 to generate three new feature maps with size reduced by a half.

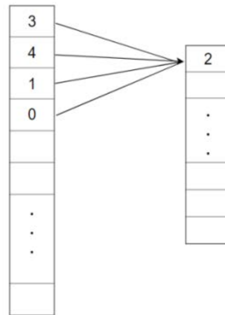


Figure 2.6. Average pooling.

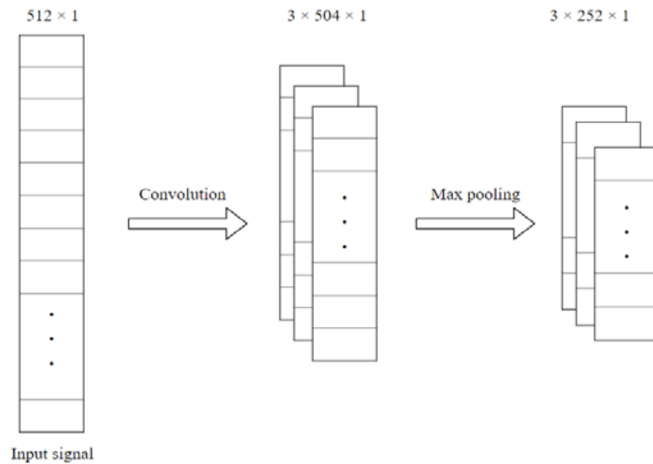


Figure 2.7. Size changes when a convolutional layer is combined with a pooling layer.

2.2.4 Fully connected layers and output layers

A fully connected layer is a conventional layer in neural network where each node is connected to *all* features generated from the preceding layer and these features are combined with weights and bias that are distinctly assigned to that node. Figure 2.8 shows a network architecture with a fully connected layer which has five neurons. In this example, the result of the fully connected layer is a vector of size 5×1. In this architecture, the connection between the last feature map and fully connected layers have $3 \times 252 \times 5 = 3780$ weights. Evidently, the complexity of the neural network is greatly impacted by the dimension reduction of the feature maps in the layers preceding to fully connected layers.

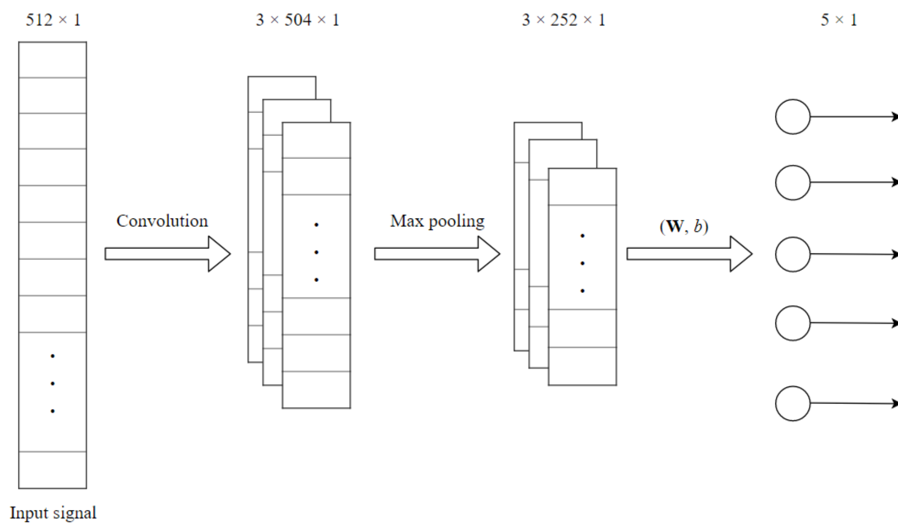


Figure 2.8. A CNN with a fully connected layer.

The outputs of the last fully connected layer are fed into the output layer to produce a vector ***s*** that predicts *class probability* of the input. Typically, *softmax* is used in the output layer where the *j*th component of ***s*** predicts the conditional probability of input ***x*** belongs to class *j* and assumes the form

$$s_j = \frac{e^{a_j}}{\sum_{k=1}^{T} e^{a_k}}$$

where $\{a_i, i = 1, 2, ..., T\}$ denote the outputs from the last fully connected layer.

2.3 Techniques Specifically Designed for CNNs

A variety of techniques have been developed over the years for performance improvement of CNNs. Below we describe three of them, namely batch normalization, shortcut, and dropout, which have been employed in our model.

2.3.1 Batch normalization

Training a network is complicated because it involves a tuning process for the parameters in the entire network using an algorithm called backpropagation for evaluating the gradient of a pre-defined loss function. In a CNN with large number of layers, the network's stability problem may arise in the sense of having either a vanishing or exploding gradient. Other difficulties facing the training process include sensitivity to network's initialization and nonlinearity saturation [23]. Batch normalization was proposed deal with some of these problems.

A batch normalization layer is always placed next to a convolutional layer, where the data flow from previous layer is normalized, scaled, and shifted mini-batch by mini-batch [21]. Table 2.1 summarizes the steps of batch normalization.

Table 2.1 Batch Normalization Steps [23]

Input: a mini-batch of m data samples $B = \{x_1, x_2, ..., x_m\}$; parameters to be leaned: γ, β

Output: normalized, scaled, and shifted *m* data samples, $\{y_i = \text{BN}_{\gamma,\beta}(x_i), i = 1, 2, ..., m\}$.

Step 1. Compute mean of $B: \mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$

Step 2. Compute variance of $B: \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$

Step 3. Normalize $B: \widehat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$ for $i = 1, 2, \ldots, m$

Step 4. Scaling and shifting: $y_i = \gamma x_i + \beta \triangleq BN_{\gamma,\beta}(x_i)$ for $i = 1, 2, \ldots, m$

From first three steps in this table, we see that each mini-batch of data samples are normalized to have zero mean and unity variance. Parameters γ, β in Step 4 are to be learned to shift and scale the normalized data properly. The work [23] shows that these two parameters are helpful to stabilized network's performance during the training process.

2.3.2 Shortcut

Over the past years, the depth of CNNs has increased significantly, from the well-known LeNet-5 with 5 layers to some recently-proposed CNN architectures with more than 150 layers [24]. As expected, however, deeper neural networks are more difficult to train as the increase of networks' depth often lead to saturated accuracy, even performance degradation. Figure 2.9 show a typical example of this type of degradation, where a 20-layer and 56-layer plain networks were tested on CIFAR-10 dataset, and the deeper network produced higher training as well test errors. Interestingly, the degradation was not caused by overfitting, but the training error increases when more layers are added to a seemingly adequate model [24].

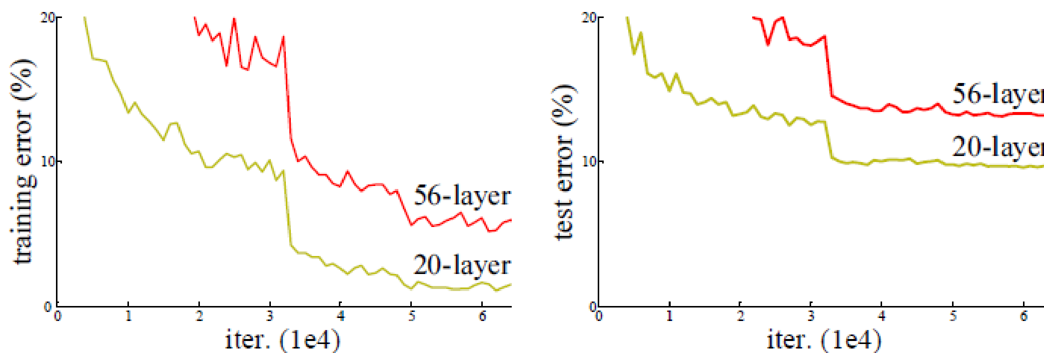


Figure 2.9. Degradation of deeper networks [24].

A deep residual learning framework is introduced in [24] to address the problem. For illustration a residual block is shown in Figure 2.10, where instead of directly feeding the output of a layer to

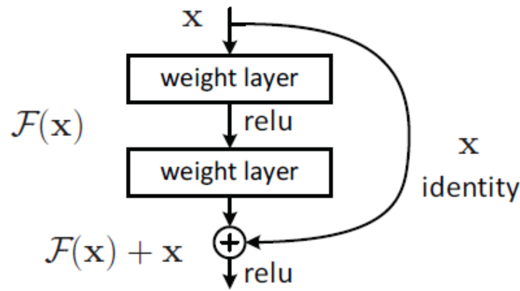
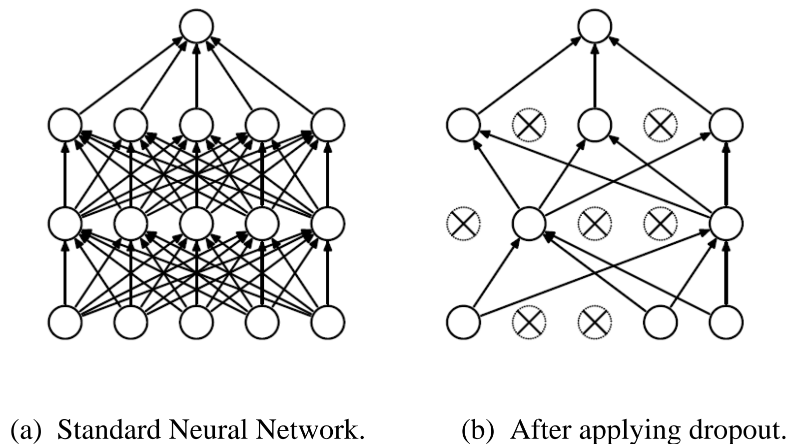


Figure 2.10. Residual learning: a building block [24].

the next layer, a *shortcut* path is added to transfer the data from previous layer(s) to current activation layer. This kind of architecture is shown to be helpful in reducing performance degradation in deeper networks [24].

2.3.3 Dropout

Dropout is a technique for addressing overfitting, which is often a serious problem in deep neural networks with a large number of parameters especially when training data is limited. A simple yet effective way to deal with overfitting problems is to terminate the training as soon as the performance on a validation set starts to get worse. Another useful technique is to introduce weight penalties of various kinds such as L_1 or L_2 regularization and soft weight sharing. However, these techniques become less effective on large-scale networks [25]. *Dropout* is proposed to address overfitting problem for such networks. The term "dropout" refers to dropping out units in various layers of a network. Figure 2.11 shows an example where dropout is employed.



(a) Standard Neural Network. (b) After applying dropout.

Figure 2.11. Dropout neural network [25].

In training process, for each iteration, hidden layer, and training sample, the technique temporarily removes randomly selected units and their incoming and outgoing connections from the network with an independently fixed probability p to produce a "thinned" network. Both forward and backpropagation are done only on this thinned network. At a test instance, as shown in Figure 2.12, the units in network are always present and the weights are multiplied by probability p.

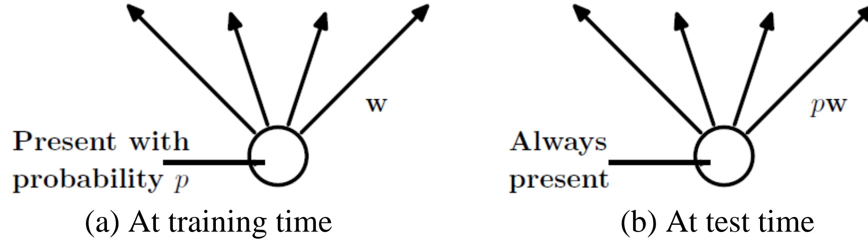


Figure 2.12. Units in dropout neural networks [25].

For better illustrating the procedure of dropout, consider a neural network with L hidden layers where the input of the lth layer is donated by $z^{(l)}$ for $l = 1, \ldots, L$, the output of the lth layer is donated by $y^{(l)}$, and $\{\boldsymbol{w}^{(l)}, b^{(l)}\}$ are weights and bias of layer l. As shown in Figure 2.13(a), in standard neural networks, the feedforward operation for any hidden unit i can described as

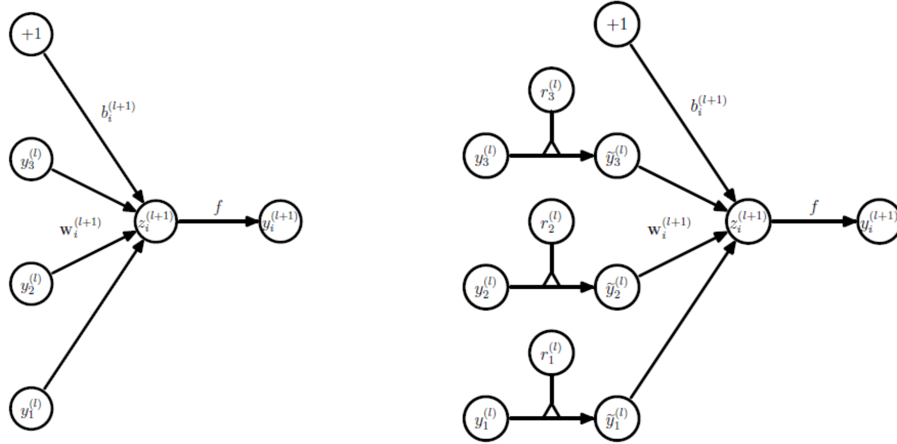
$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \boldsymbol{y}^{(l)} + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f(z_i^{(l+1)})$$

where f is an activation function.

As shown in Figure 2.13(b), in dropout networks, the feedforward operation becomes

$$r_j^{(l)} \sim \text{Bernoulli}(p)$$
$$\tilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} \odot \boldsymbol{y}^{(l)}$$
$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \tilde{\boldsymbol{y}}^{(l)} + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f(z_i^{(l+1)})$$

where $\odot$ donates element-wise product and each element in vector $\boldsymbol{r}^{(l)}$ is an independent Bernoulli-random variable which has possibility p of being 1, $\tilde{\boldsymbol{y}}^{(l)}$ is thinned output and is used as input of the nest layer. This operation is applied to each layer and produces a set of thinned sub-networks.



(a) Standard network (b) Dropout network

Figure 2.13. Comparison of the basic operations of a standard a dropout network with dropout [25].

2.4 Our Models

Inspired by the work in [8], in this section we present three CNN models and examine several related implementation issues.

2.4.1 Architecture

These three models are CNNs with 35 layers, 19 layers and 9 layers, respectively. A high-level architectural illustration of the models is shown in Figure 2.14.

A common point of these models is that each model consists of three blocks: a first block, a loop block, and a final block. The first block includes three convolutional layers, the loop block includes two convolutional layers, while the final block has a convolutional layer, a fully-connected layer, and softmax activation that predicts probability of five classes of heart diseases. In the 35-layer network, the number of loops was set to 15, while this number was set to 7 and 2 in the 19-layer and 9-layer models, respectively. Batch normalization and ReLU activation are applied prior to each convolutional layer. To prevent from overfitting, dropout layers are applied between the convolutional layers.

In addition, shortcut connections are employed in our models. These shortcut connections improve the training process by allowing convolutional layers to acquire information more directly from

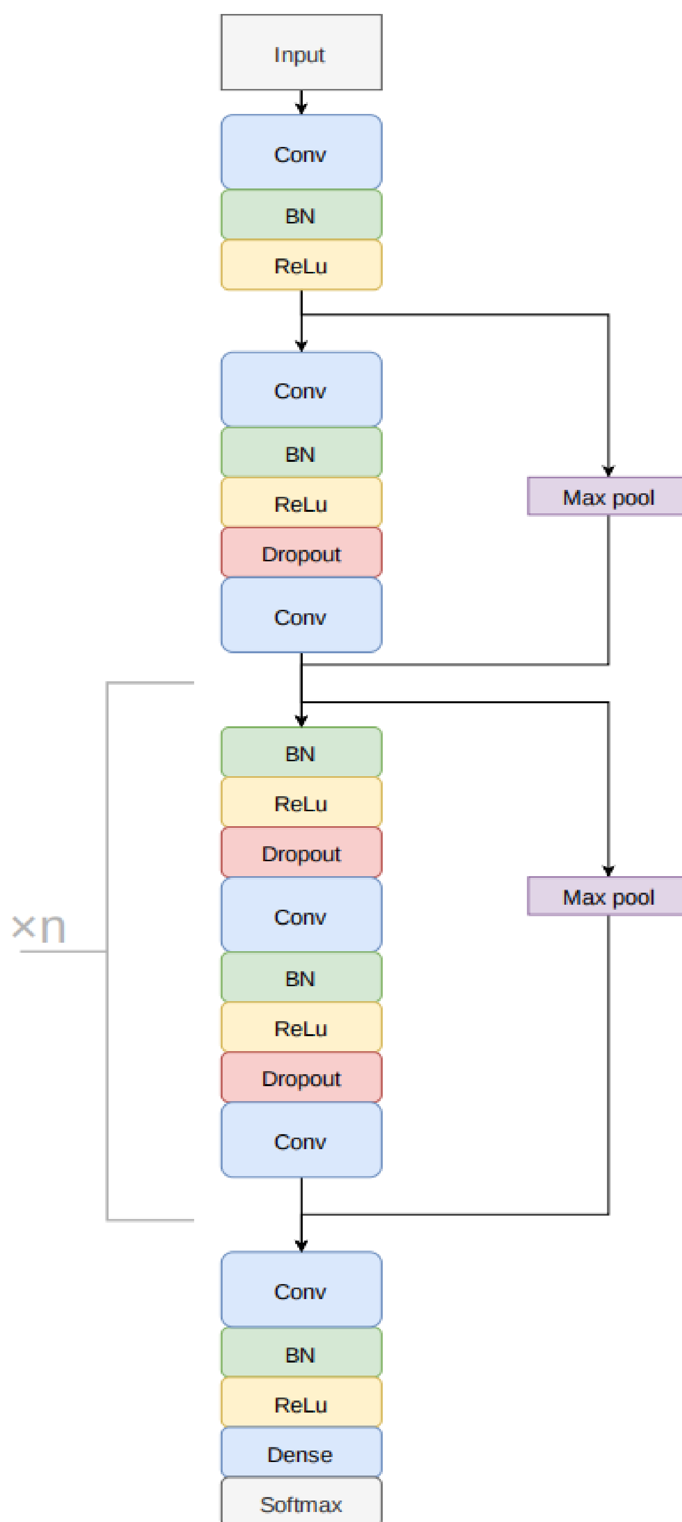


Figure 2.14. Architectural illustration of the three CNNs.

preceding layers. in our models there are $n + 1$ residual blocks, where n donates the number of loops. Max pooling operation is applied to subsample the inputs of shortcut connections. If the index of blocks is even, the size of max pooling is set to a, otherwise, it is set to b, where $a = 2$ in the 35-layer network, $a = 4$ in the 19-layer work, and $a = 16$ in 9-layer network, while $b = 1$ in the 35-layer and 19-layer networks and $b = 8$ in the 9-layer network. This operation reduces the size of shortcut connections' output to a suitable number to ensure that it can be added to the input of the layer to which it is fed.

2.4.2 Implementation

The proposed models are built and trained from scratch in Python by using Keras, a popular open-source neural-network library. The layers in the networks are implemented as follows:

Convolutional layers:
```
layer = Conv1D(filters = filter_length,
               kernel_size = config.kernel_size,
               padding = 'same',
               strides = 1,
               kernel_initializer = 'he_normal')(layer)
```

Activation layers:
```
layer = Activation('relu')(layer)
```

Batch Normalization:
```
layer = BatchNormalization()(layer)
```

Dropout layers:
```
layer = Dropout(config.drop_rate)(layer)
```

Shortcut:
```
shortcut = MaxPooling1D(
           pool_size=subsample_length)(layer)
```

The weights in convolutional layers are initialized by using he_normal method [23]. Model class and compile function in Keras were used for training networks, Adam [25] was used as the optimizer, categorical_crossentropy loss is used as loss function, the initial learning rate was set to 0.2 and was then reduced by a factor of 10 according to improvement of the validation loss as follows:

```
model = Model(inputs=inputs, outputs=outputs)
        adam = Adam(lr = 0.2, beta_1 = 0.9, beta_2 = 0.999, epsilon=None,
        decay = 0.0, amsgrad= False)
model.compile(optimizer= adam, loss='categorical_crossentropy',
metrics=['accuracy'])
```

Chapter 3 Performance Evaluation

This chapter is devoted to experimental studies of the three models described in Chap. 2. We begin by introducing the dataset from the MIT-BIH database[26] and an implementation procedure for pre-processing the dataset. We then report experimental results of our models when applied to the MIT-BIH dataset. These include numerical evidence that indicates how dropout, batch normalization, and shortcut improve performance of the models as long as the MIT-BIH dataset is concerned.

3.1 The Datasets

3.1.1 The MIT-BIH database

The MIT-BIH database provides 48 half-hour two-channel ECG recordings, which are obtained by the BIH Arrhythmia Laboratory from a set of 4000 24-hour ambulatory ECG recordings collected from patients at Boston's Beth Israel Hospital between 1975 and 1979. The datasets used in this project were derived from the MIT-BIH database and modified to Python-friendly forms using the algorithm in [27]. The modified datasets contain the four parts described in Table 3.1.

Table 3.1 Datasets Description

Name	Columns	Rows	Description
`normalrr1.mat`	33617	553	Normal beat
`apcrr2.mat`	2178	553	Atrial premature beat
`pvcrr2.mat`	5725	553	Premature ventricular contraction
`lbbbrr1.mat`	3992	553	Left bundle branch block beat
`rbbbrr1.mat`	4467	553	Right bundle branch block beat

The datasets contain five classes of heart beats, namely normal beat, atrial premature beat, premature ventricular contraction, left bundle branch block beat and right bundle branch block

beat. Each beat is represented by a vector of size 553 by 1, so in total there are 49979 beats in the datasets.

3.1.2 Data Preparation

In order to satisfy the requirements of Keras, several data-preparation operations are needed before training. These operations are implemented in Python using libraries such as **numpy**, **scipy** and **scikits-learn**. For clarity key Python commands involved in each operation are presented. For simplicity, in the rest of the chapter terms function(s) and command(s) are referred to as Python function(s) and command(s), respectively.

Using function **loadmat** in **scipy** directly, five datasets (see Table 3.1) were extracted from **.mat** files as follows:

```
data = scio.loadmat(datafile)
_, _, _,  key = data.keys()
```

Since the extracted datasets do not have labels indicating to which class the data belong, labels are assigned to each dataset as

```
m,n = np.shape(data)
label = np.zeros(shape = [n, nclasses], dtype = float)
label[:, label_class] = 1
```

Variable **label** is a matrix of size **n** by **nclasses**, where **n** denotes height of the dataset and **nclasses** denotes number of the classes in the dataset. In this project, **n** = 512 and **nclasses** = 5.

The class labels are *one-hot* encoded. This is to say, for a data point (i.e., a heart-beat waveform) belonging to class *i*, its label is a 5 by 1 vector with *i*th component set to value 1 and other components set to value 0.

By using **concatenate** function in **numpy**, we obtain a single variable **x** containing all data from the dataset, and a variable **y** containing the corresponding labels of **x**:

```
n_data, n_label = loaddata(0, 'dataset/normalrr1.mat')
a_data, a_label = loaddata(1, 'dataset/apcrr2.mat')
x = np.concatenate((x, a_data))
y = np.concatenate((y, a_label))
p_data, p_label = loaddata(2, 'dataset/pvcrr2.mat')
x = np.concatenate((x, p_data))
```

```
y = np.concatenate((y, p_label))
r_data, r_label = loaddata(3, 'dataset/rbbbrr1.mat')
x = np.concatenate((x, r_data))
y = np.concatenate((y, r_label))
l_data, l_label = loaddata(4, 'dataset/lbbbrr1.mat')
x = np.concatenate((x, l_data))
y = np.concatenate((y, l_label))
```

By using function `train_test_split` from `scikits-learn`, we obtain 20 different training sets and one test set:

```
train_x, train_y = dict(), dict()
for i in range(20):
    train_x[i], x_val, train_y[i], y_val = train_test_split(x, y,
test_size=0.2, random_state=None)
```

where `train_x` and `train_y` are dictionaries containing all training data and their labels, while `x_val` and `y_val` contain test data and their labels.

3.2 Experimental Results

3.2.1 Three CNNs based models

By using function `fit` in `Model` class, our three models were trained and tested using the preprocessed dataset, where the number of training epochs was set to 50 and the batch size was set to 128:

```
model.fit(xtr, ytr,
          validation_data=(xte, yte),
          epochs=config.epochs,
          batch_size=config.batch,
          callbacks=callbacks,
          initial_epoch=initial_epoch)
```

Since the MIT-BIH database doesn't provide separate testing data, we select at random a part of the data for training purposes and another (non-overlapping) part of the data for testing. A point to note is that whether or not a specific heart-beat waveform is used as training or testing is entirely determined by the initial random state used for the random partition. As a result, the numerical findings based on such data sets will very likely vary when one uses data sets constructed using a

different initial random state. In our study, a total of 20 randomly selected (with distinct initial states) data sets were examined and the results are statistically analyzed in terms of classification accuracy, training time, and testing time.

In this report, the classification accuracy is defined as the ratio of the number of samples which are predicted correctly to the total number of samples:

$$classification\ accuracy = \frac{the\ number\ of\ correctly\ predict\ samples}{total\ number\ of\ samples}$$

The experiments were carried out on `Google Collaboratory`, and the results obtained are reported in Tables 3.2 and 3.3.

Table 3.2 Classification accuracy of three CNN-based models

Training set	Classification accuracy		
	9-layer network	19-layer network	35-layer network
1	0.9867	0.9882	0.9943
2	0.9876	0.9901	0.9886
3	0.9887	0.9923	0.9895
4	0.9877	0.9911	0.9888
5	0.9901	0.9931	0.9865
6	0.9845	0.9930	0.9903
7	0.9845	0.9884	0.9889
8	0.9876	0.9913	0.9882
9	0.9887	0.9896	0.9932
10	0.9823	0.9928	0.9901
11	0.9865	0.9921	0.9884
12	0.9812	0.9913	0.9895

13	0.9876	0.9987	0.9927
14	0.9888	0.9902	0.9927
15	0.9876	0.9923	0.9921
16	0.9834	0.9889	0.9897
17	0.9898	0.9923	0.9896
18	0.9902	0.9886	0.9890
19	0.9884	0.9885	0.9920
20	0.9854	0.9912	0.9883
Average	0.9869	0.9910	0.9900

Table 3.3 Training and testing time of three CNN-based models

Number of training set	Training time(s)			Testing time per beat(ms)		
	9-layer network	19-layer network	35-layer network	9-layer network	19-layer network	35-layer network
1	2019	2411	5698	0.13	0.17	0.22
2	2087	2507	5709	0.15	0.17	0.21
3	1787	2525	5712	0.14	0.18	0.22
4	2098	2514	5723	0.13	0.17	0.20
5	2088	2614	5698	0.14	0.13	0.26
6	2386	2603	5689	0.13	0.14	0.15
7	1674	1675	5702	0.13	0.15	0.22
8	1968	2506	5713	0.14	0.14	0.24

9	2098	2634	5709	0.16	0.15	0.22
10	2136	2678	5767	0.13	0.17	0.23
11	2237	1878	5714	0.18	0.17	0.25
12	2076	2609	5718	0.15	0.13	0.14
13	2037	2567	5723	0.17	0.13	0.31
14	1765	2656	5732	0.14	0.13	0.27
15	2056	2768	5721	0.14	0.15	0.23
16	2123	2767	5767	0.15	0.17	0.23
17	2098	2509	5702	0.14	0.16	0.26
18	1476	2512	5709	0.17	0.16	0.20
19	2076	2607	5712	0.13	0.16	0.24
20	2020	2687	5721	0.09	0.14	0.22
Average	2015	2511	5717	0.14	0.15	0.23

From Tables 3.2, it is observed that the 19-layer network was able to achieve 99.1% accuracy in average. Also note that the 35-layer and 19-layer architectures provide almost the same prediction accuracy; but from Table 3.3 the 19-layer model is considerably faster than the one with 35-layers, in terms of both the training and testing times required.

Figure 3.1 shows the confusion matrices of the three models that were averaged over the 20 datasets. It is observed that the distribution of the misclassifications appears to be somewhat even, with more misclassifications occurred for classes "*n*" and "*a*" relative to classes "*l*" and "*r*".

	n	a	p	r	l
n	3318	13	9	2	1
a	10	186	0	1	0
p	13	7	578	1	1
r	0	0	1	453	0
l	3	1	2	0	398

(a) the 9-layer model

	n	a	p	r	l
n	3328	7	10	1	0
a	9	197	1	0	0
p	7	2	577	1	3
r	0	1	1	455	1
l	0	0	1	0	396

(b) the 19-layer model

	n	a	p	r	l
n	3326	8	3	1	0
a	10	195	2	1	0
p	6	1	576	1	3
r	2	3	8	454	0
l	0	0	1	0	397

(c) the 35-layer model

Figure 3.1. Confusion matrices averaged over 20 datasets.

To examine the robustness of the CNN-based neural nets against variations in the relative sizes of training and testing data, we define *splitting ratio* as ratio of the size of testing set to the size of the whole dataset, and evaluate the average performance of the three models on splitting ratios ranging from 0.1 to 0.5. The numerical results are shown in Table 3.4.

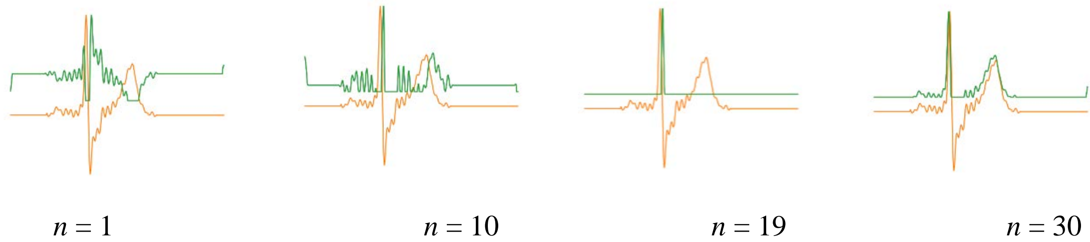
Table 3.4 Classification accuracy versus splitting ratio

Splitting ratio	9-layer network	19-layer network	35-layer network
0.1	0.9869	0.9910	0.9900
0.2	0.9863	0.9902	0.9895
0.3	0.9856	0.9875	0.9876
0.4	0.9821	0.9868	0.9862
0.5	0.9813	0.9865	0.9858

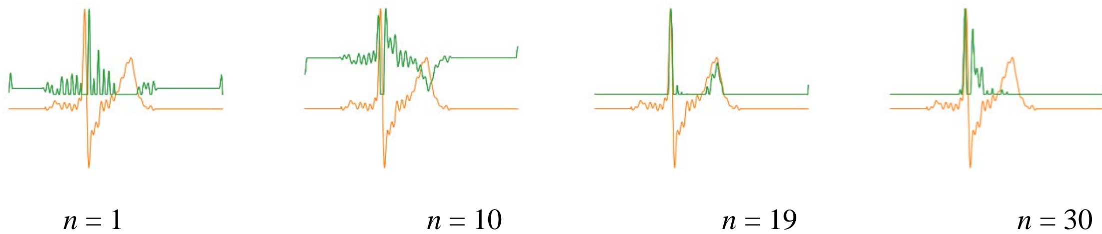
From Table 3.4, it is observed that the three models are rather insensitive to the changes in splitting ratio.

3.2.2 Feature maps

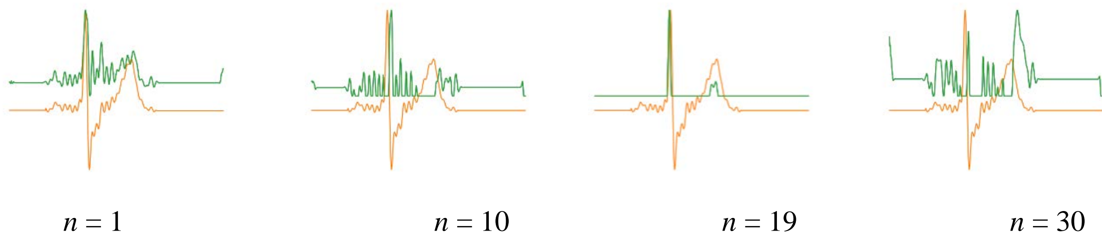
An important characteristic of CNN networks is that each convolutional layer (C-layer) generates several feature maps that are carried forward to the next C-layer through sub-sampling and pooling. It is therefore of interest to visualize individual features from different channels at different convolutional layers. Given a input ECG signal, each row of Figures 3.2 to 3.4 depicts 4 waveforms in green color from the *n*th feature maps produced by one of the first three C-layers of a given trained model, where the waveform of the input signal is shown in orange was for reference purpose.



(a) Feature maps of activation layer 1.

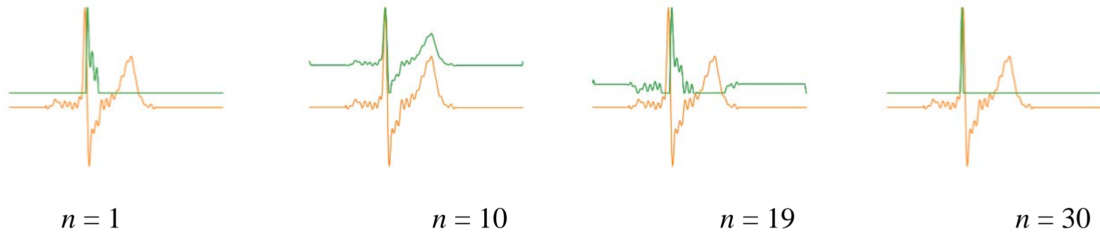


(b) Feature maps of activation layer 2.



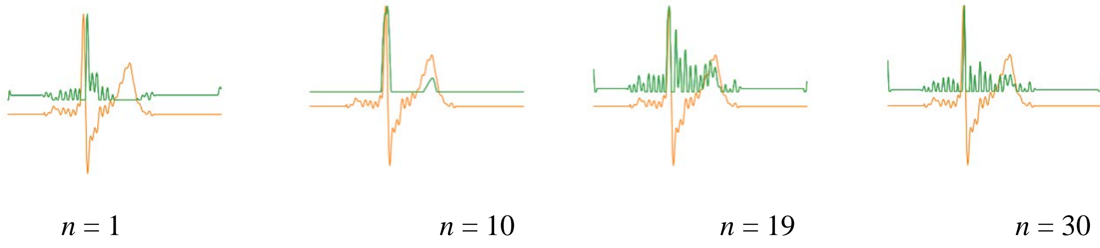
(c) Feature maps of activation layer 3.

Figure 3.2. The *n*th feature maps in the 9-layer network.



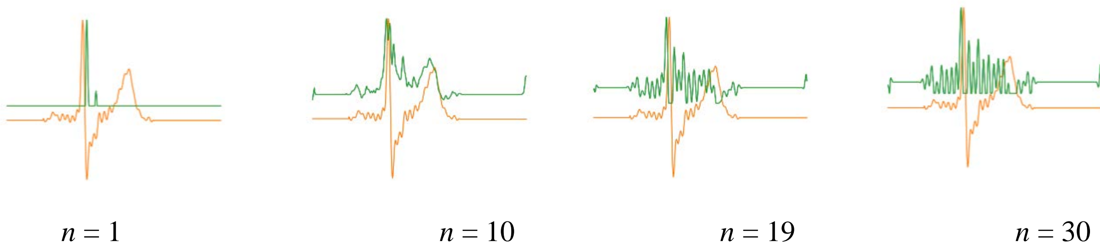
$n = 1$ $n = 10$ $n = 19$ $n = 30$

(a) Feature maps of activation layer 1.



$n = 1$ $n = 10$ $n = 19$ $n = 30$

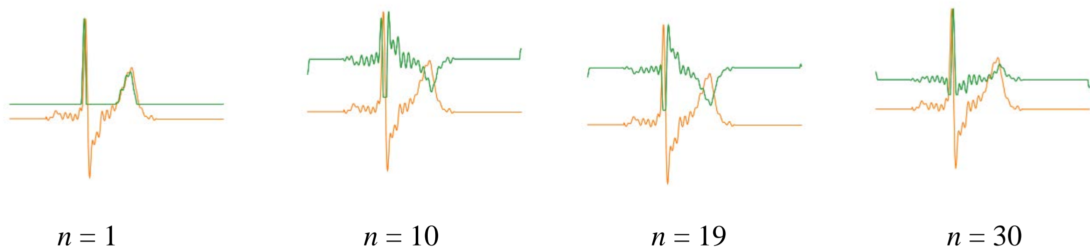
(b) Feature maps of activation layer 2.



$n = 1$ $n = 10$ $n = 19$ $n = 30$

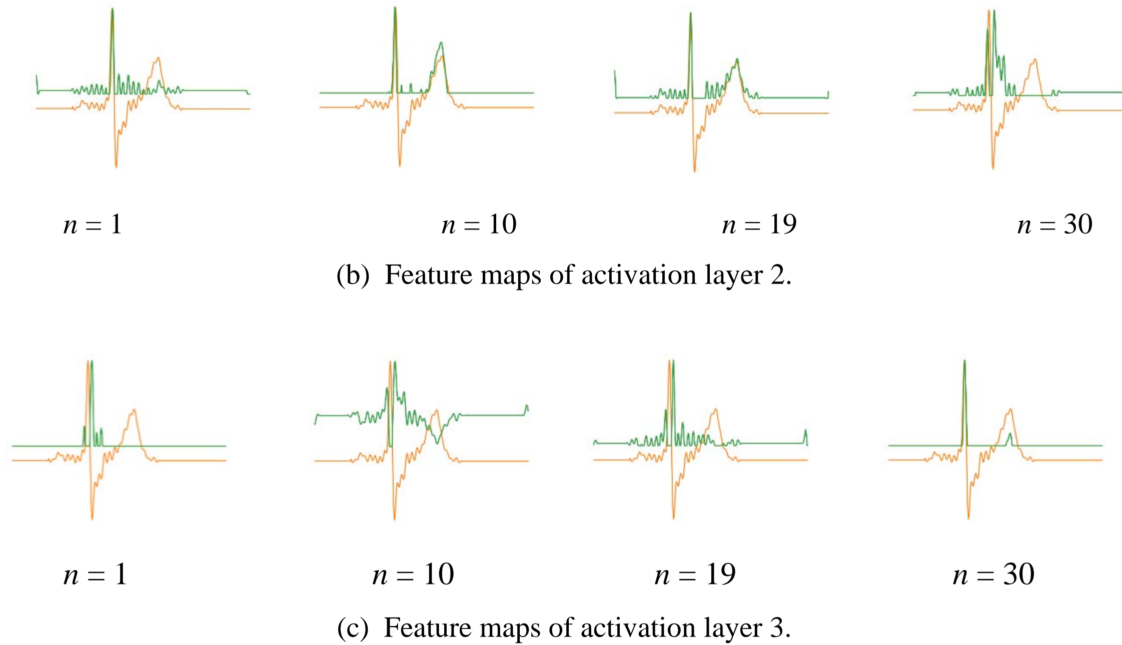
(c) Feature maps of activation layer 3.

Figure 3.3. The *n*th feature maps in the 19-layer network



$n = 1$ $n = 10$ $n = 19$ $n = 30$

(a) Feature maps of activation layer 1.



(b) Feature maps of activation layer 2.

(c) Feature maps of activation layer 3.

Figure 3.4. The *n*th feature maps in the 35-layer network.

The waveforms from different feature maps in different layers appear to be somehow related to the input ECG signal, especially its P wave, QRS complex, and T wave, but only in an abstract way. This is not at all surprising because feature maps produced from deep CNN networks have been known to be hard to interpret [19].

3.2.3 Architectures without batch normalization

For all three aforementioned models, performance difference with and without batch normalization are examined. The architecture of the three CNN models without batch normalization layers is illustrated in Fig. 3.5. Classification accuracy and training/testing time of the three models without batch normalization are reported in Tables 3.5 and 3.6, respectively. On comparing with the results from Tables 3.2 and 3.3 for networks with batch normalization, it is observed that without batch normalization the training and testing time were reduced about 30-50%, but the networks were difficult to train, and after 50 training epochs the prediction accuracy became saturated at an accuracy level below 70% which was considerably inferior to their counterparts equipped with batch normalization.

Table 3.5 Classification accuracy of three CNNs without batch normalization

Number of training set	9-layer network		19-layer network		35-layer network	
	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy
1	0.6728	0.6691	0.6643	0.6590	0.6577	0.6531
2	0.6733	0.6689	0.6598	0.658	0.6563	0.6534
3	0.6733	0.6702	0.6575	0.6540	0.6594	0.6570
4	0.6718	0.6710	0.6604	0.6551	0.6619	0.6565
5	0.6722	0.6721	0.6493	0.6431	0.6612	0.6563
6	0.6647	0.6602	0.6573	0.6540	0.6607	0.6544
7	0.6700	0.6678	0.6484	0.6442	0.6477	0.6454
8	0.6711	0.6653	0.6617	0.6558	0.6599	0.6554
9	0.6834	0.6771	0.6542	0.6534	0.6554	0.6553
10	0.6687	0.6677	0.6560	0.6548	0.6572	0.6570
11	0.6601	0.6540	0.6593	0.6578	0.6595	0.6554
12	0.6703	0.6666	0.6596	0.6540	0.6560	0.6554
13	0.6621	0.6600	0.6695	0.6671	0.6570	0.6557
14	0.6798	0.6780	0.6618	0.6555	0.6566	0.6559
15	0.6725	0.6690	0.6696	0.6643	0.6590	0.6567
16	0.6776	0.6734	0.6510	0.6489	0.6597	0.6541
17	0.6567	0.6554	0.6514	0.6498	0.6482	0.6437
18	0.6608	0.6545	0.6515	0.6478	0.6477	0.6442

19	0.6811	0.6812	0.6518	0.649	0.6564	0.6555
20	0.6707	0.6877	0.6576	0.6589	0.6567	0.6412
Average	0.6728	0.6684	0.6576	0.6542	0.6567	0.6531

Table 3.6 Training and testing time of three CNNs without batch normalization

Number of training set	Training time(s)			Testing time per beat(ms)		
	9-layer network	19-layer network	35-layer network	9-layer network	19-layer network	35-layer network
1	606	1315	2346	0.039	0.093	0.091
2	626	1368	2351	0.045	0.093	0.086
3	536	1378	2352	0.042	0.098	0.091
4	629	1372	2357	0.039	0.093	0.082
5	626	1426	2346	0.042	0.071	0.107
6	716	1420	2343	0.039	0.076	0.062
7	502	914	2348	0.039	0.082	0.091
8	590	1367	2352	0.042	0.076	0.099
9	629	1437	2351	0.048	0.082	0.091
10	641	1461	2375	0.039	0.093	0.095
11	671	1025	2353	0.054	0.093	0.103
12	623	1423	2355	0.045	0.071	0.058
13	611	1400	2357	0.051	0.071	0.128
14	530	1449	2360	0.042	0.071	0.111

15	617	1510	2356	0.042	0.082	0.095
16	637	1510	2375	0.045	0.093	0.095
17	629	1369	2348	0.042	0.087	0.107
18	443	1370	2351	0.051	0.087	0.082
19	623	1422	2352	0.039	0.087	0.099
20	606	1466	2356	0.027	0.076	0.091
Average	605	1370	2354	0.042	0.082	0.095

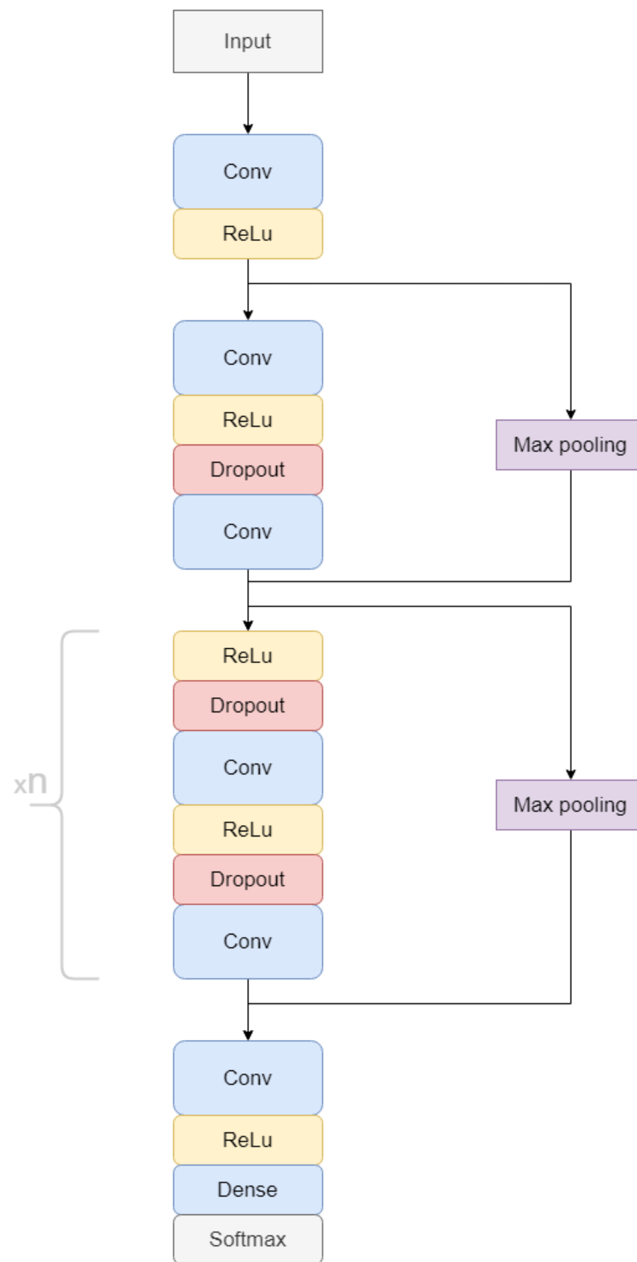


Figure 3.5. Architectural illustration of the three CNNs without batch normalization.

3.2.4 Networks without shortcut

Shortcut has demonstrated impressive results, especially for very deep neural networks like the ResNet-152 [24]. However, the CNN models considered in this project are much shallower with no more than 35 layers. So it remains interesting to evaluate our models' performance with and

without shortcut, especially for the 9-layer and 19-layer networks. To find an answer, we removed all shortcut from the three models as shown in Figure 3.6. From this figure, we can see that $n + 1$ addition operations and max pooing operations are removed, where n donates the number of loops. The classification accuracy and training/testing time of the networks without shortcut are given in Tables 3.7 and 3.8, respectively. We see that in average the accuracy of the 9-layer and 19-layer networks achieved 98%. On comparing with their counterparts with shortcut, the accuracy achieved by the 9-layer network remains practically the same, while there is a 0.8% reduction in accuracy for the 19-layer network. Considerable performance degradation occurs for the 35-layer network, this network was trained in 50 epochs, yet classification accuracy remains below 70%. The results of this instance clearly shows that shortcut can be very effective in making neural networks relatively easier to train, especially for deep neural networks. On comparing to their counterparts with shortcut, the training and testing time is reduced by about 5%. Considering we only removed $n + 1$ additions and max pooling, this outcome appears to be a reasonable one.

Table 3.7 Classification accuracy of three models without shortcut

Number of training set	9-layer network		19-layer network		35-layer network	
	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy
1	0.9819	0.9798	0.9855	0.9811	0.6742	0.6701
2	0.9853	0.9827	0.9822	0.9792	0.6741	0.6708
3	0.9854	0.9838	0.9832	0.9807	0.6783	0.6749
4	0.9842	0.9828	0.9830	0.9811	0.6773	0.6741
5	0.9862	0.9852	0.9851	0.9834	0.6775	0.6739
6	0.9851	0.9797	0.9900	0.9843	0.6753	0.6719
7	0.9836	0.9797	0.9828	0.9802	0.6664	0.6630
8	0.9864	0.9828	0.9869	0.9835	0.6767	0.6725

9	0.9847	0.9838	0.9820	0.9815	0.6772	0.6730
10	0.9879	0.9875	0.9900	0.9847	0.6778	0.6744
11	0.9872	0.9817	0.9854	0.9841	0.6765	0.6726
12	0.9864	0.9824	0.9889	0.9832	0.6775	0.6737
13	0.9938	0.9827	0.9901	0.9886	0.6769	0.6736
14	0.9853	0.9839	0.9822	0.9808	0.6768	0.6736
15	0.9874	0.9827	0.9842	0.9840	0.6781	0.6741
16	0.9840	0.9786	0.9819	0.9808	0.6758	0.6719
17	0.9874	0.9849	0.9853	0.9832	0.6642	0.6607
18	0.9853	0.9837	0.9844	0.9805	0.6642	0.6609
19	0.9837	0.9836	0.9833	0.9803	0.6769	0.6734
20	0.9863	0.9845	0.9831	0.9816	0.6622	0.6585
Average	0.9859	0.9828	0.9849	0.9823	0.6742	0.6706

Table 3.8 Training and testing time of three models without batch normalization

Number of training set	Training time(s)			Testing time per beat(ms)		
	9-layer network	19-layer network	35-layer network	9-layer network	19-layer network	35-layer network
1	1908	2210	5305	0.12	0.16	0.20
2	1973	2298	5315	0.14	0.16	0.20
3	1689	2315	5318	0.13	0.17	0.20
4	1983	2305	5328	0.12	0.16	0.19

5	1974	2396	5305	0.13	0.12	0.24
6	2255	2386	5297	0.12	0.13	0.14
7	1582	1535	5309	0.12	0.14	0.20
8	1860	2297	5319	0.13	0.13	0.22
9	1983	2415	5315	0.15	0.14	0.20
10	2019	2455	5369	0.12	0.16	0.21
11	2114	1722	5320	0.17	0.16	0.23
12	1962	2392	5324	0.14	0.12	0.13
13	1925	2353	5328	0.16	0.12	0.29
14	1668	2435	5337	0.13	0.12	0.25
15	1943	2537	5326	0.13	0.14	0.21
16	2007	2536	5369	0.14	0.16	0.21
17	1983	2300	5309	0.13	0.15	0.24
18	1395	2303	5315	0.16	0.15	0.19
19	1962	2390	5318	0.12	0.15	0.22
20	1909	2463	5326	0.09	0.13	0.20
Average	1905	2302	5323	0.13	0.14	0.21

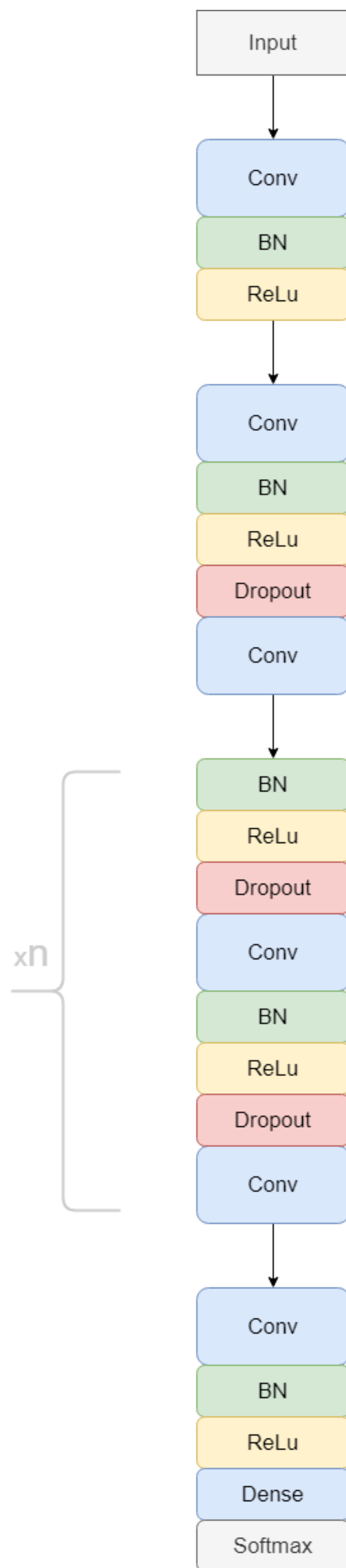


Figure 3.6. Architectural illustration of the three CNNs without shortcut.

3.2.5 Networks with different dropping rate

In this section, we test our models with different dropping rate in order to explore the effect of dropout on the three networks. Table 3.9 shows average training accuracy for 20 training sets and testing accuracy of the three networks after 50 training epochs with dropping rate varying from 0 to 0.9.

Table 3.9 Average training and testing accuracy of CNNs with different dropping rates

Dropping rate	9-layer network		19-layer network		35-layer network	
	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy	Training accuracy	Testing accuracy
0	0.9858	0.9640	0.9944	0.9768	0.9969	0.9784
0.1	0.9887	0.9869	0.9851	0.9840	0.9928	0.9850
0.2	0.9874	0.9713	0.9914	0.9892	0.9973	0.9900
0.3	0.9798	0.9867	0.9923	0.9910	0.9934	0.989
0.4	0.9737	0.9611	0.9887	0.9850	0.9832	0.9824
0.5	0.9708	0.9623	0.987	0.9847	0.9885	0.9854
0.6	0.9749	0.9748	0.9834	0.9822	0.9801	0.9786
0.7	0.9314	0.8749	0.9667	0.9127	0.9554	0.8279
0.8	0.8976	0.7645	0.8849	0.7451	0.8801	0.7631
0.9	0.8342	0.6153	0.8937	0.6117	0.8254	0.6691

From the table, it is observed that when the dropping rate is set to 0, overfitting occurs in the three architectures. The best accuracies are achieved when the dropping rate is set to 0.1 for the 9-layer network, 0.3 for the 19-layer network, and 0.2 for the 35-layer network. We also remark that increasing dropping rate tends to make the training more difficult, and hence degraded testing accuracy. This is because removing too many units in dropout layers can have an impact on the networks' performance in a negative manner.

3.2.6 Comparisons with other techniques

Table 3.10 summarizes the performance of several ECG-based classifiers achieved using the same datasets from MIT-BIH Arrhythmia database. It is observed that most of the methods are concerned with the classification of the classes heart diseases as we do. Yang et al. [28] obtained an accuracy of 98.10% by using PCAnet and linearSVM. CNN was used in [29, 30] and accuracy of 94.03% and 98.10% were achieved, respectively. The works in [31] used LSTM and they obtained impressive accuracy. We also remark that the work in [32] focuses on recognition of a total of 17 classes, where the same 1D-CNN was used as we did to achieve an accuracy of 91.33%. On comparing to these methods, we see that the CNN models studied in this project have demonstrated competitive classification accuracy.

The method proposed in reference [31] demonstrated a slightly better performance than what our models did. In [31], BLSTMs is used and a wavelet sequence (WS) layer is placed in front of the BLSTMs. If they would use the same 553 by 1 input data as we do, the total number of parameters involved in this architecture would be

$$
\begin{aligned}
&(n \times L)_{the\ WS\ layer} + 4(n \times m_1 + n^2 + n)_{the\ 1st\ BLSTM\ layer} + 4(m_1 \times m_2 + m_1^2 + m_1)_{the\ 2nd\ BLSTM\ layer} + (n_d + n_y) \\
&= (553 \times 3) + 4(553 \times 64 + 553^2 + 553) + 4(64 \times 32 + 32^2 + 32) + (128 \times 5) \\
&= 1,381,731
\end{aligned}
$$

where n donates the length of the input signal, L donates the number of levels in the WS layers, m_1 donates the number of units in the first BLSTM layer, m_2 donates the number of units in the second BLSTM layer, n_d donates the number of units in the first dense layer and n_y donates the number of units in the second dense layer.

On comparison with the method in [31], the parameters involved in our 19-layer networks is

$$
\begin{aligned}
&(l+1) \times n_{c1\,the\ first\ CNN\ layer} + 10 \times (n_{c1} \times l + 1) n_{c1\,CNN\ layers\ No.2\text{-}11} + (n_{c1} \times l + 1) \times n_{c2\,CNN\ layer\ No.12} \\
&+ 6 \times (n_{c2} \times l + 1) n_{c2\,CNN\ layer\ No.13\text{-}18} + (n_{c2} \times n_d)_{dense\ layer} \\
&= (9+1) \times 32 + 10 \times (32 \times 9 + 1) + (32 \times 9 + 1) \times 64 + 6 \times (64 \times 9 + 1) \times 64 + (64 \times 5) \\
&= 333,184
\end{aligned}
$$

where l donates the length of kernel, n_{c1} donates the number of kernels in the first convolutional layer to the 11th convolutional layers, n_{c2} donates the number of kernels in the 12th convolutional layer to the 18th convolutional layer and n_d donates the number of units in dense layers.

We can see that the number of parameters involved in our 19-layer networks is much less that of the architecture in [31], which means that our model can be trained faster.

Table 3.10 Performance of several classifiers using the same database

Network	Year	No. of classes	Feature set	Classifier	Accuracy
Acharya et al.[29]	2017	5	Raw data	CNN	94.03%
Yang et al. [28]	2018	5	PCAnet	Linear SVM	97.94%
Oh et al [30].	2018	5	Raw data	CNN-LSTM	98.10%
Yildirim [31]	2018	5	Raw data	DBLSTM-WS3	99.39%
Yildirim et al.[32]	2018	17	Raw data	1D-CNN	91.33%
Our work	2020	5	Raw data	1D-CNN	99.10%

Chapter 4 Conclusions

In this report, we have presented three CNN-based models for the classification of ECG waveforms. These models have been tested on the MIT-BIH datasets and the effects of several techniques including batch normalization, dropout, and shortcut have been examined. The report also includes performance comparisons of the CNN models with several other classification methods, and the CNN-based models have shown some advantages.

As pointed out in [11], "*A key advantage of neural networks over traditional machine learning is that the former provides a higher-level abstraction of expressing semantic insights about data domains by architectural design choices in the computational graph. The second advantage is that neural networks provide a simple way to adjust the complexity of a model by adding or removing neurons from the architecture according to the availability of training data or computational power. A large part of the recent success of neural networks is explained by the fact that the increased data availability and computational power of modern computers has outgrown the limits of traditional machine learning algorithms, which fail to take full advantage of what is now possible. The performance of traditional machine learning remains better at times for smaller data sets because of more choices, greater ease of model interpretation, and the tendency to handcraft interpretable features that incorporate domain-specific insights.*

With limited data, the best of a very wide diversity of models in machine learning will usually perform better than a single class of models (like neural networks). This is one reason why the potential of neural networks was not realized in the early years. The "big data" era has been enabled by the advances in data collection technology; virtually everything we do today, including purchasing an item, using the phone, or clicking on a site, is collected and stored somewhere."

On the other hand, however, the datasets used in this report is relatively small. There are 20 classes of heart beats in the MIT-BIH datasets, but only five of them were used in our studies. Large scale databases such as those in [8] are about 80 times greater than ours and it is rational as future work to conduct evaluation of CNN-based models on these new datasets. In addition, for performance improvement future work shall consider integration of techniques like LSTM, which have demonstrated impressive abilities for CNN architectures [29], [30].

Appendix Python Codes

A. Code for Configuration

```
import argparse

parser = argparse.ArgumentParser()

def add_argument_group(name):
    arg = parser.add_argument_group(name)
    return arg

misc_arg = add_argument_group('misc')
misc_arg.add_argument('--input_size', type=int, default = 512)

graph_arg = add_argument_group('graph')
graph_arg.add_argument('--filter_length', type=int, default = 32)
graph_arg.add_argument('--kernel_size', type=int, default = 9)
graph_arg.add_argument('--drop_rate', type=float, default = 0.3)

train_arg = add_argument_group('train')
train_arg.add_argument('--epochs', type=int, default = 20)
train_arg.add_argument('--batch', type=int, default = 128)
train_arg.add_argument('--patience', type=int, default = 80)
train_arg.add_argument('--min_lr', type=float, default = 0.00005)
train_arg.add_argument('--checkpoint_path', type=str, default = None)
train_arg.add_argument('--resume_epoch', type=int)
train_arg.add_argument('--ensemble', type=bool, default = False)

def get_config():
    config, unparsed = parser.parse_known_args()
    return config
```

B. Code for 9-layer Network

```
from __future__ import division, print_function

from keras import backend as K
```

```python
from keras.layers import Input, Conv1D, Dense, add, Dropout, MaxPooling1D,
Activation, BatchNormalization, \
    Lambda
from keras.models import Model
from keras.optimizers import Adam

from config import get_config


def ECG_model(config):
    """
    reference to codes at
    https://github.com/awni/ecg/blob/master/ecg/network.py
    and
    https://github.com/physhik/ecg-mit-bih/blob/master/src/graph.py
    """


    def first_conv_block(inputs, config):
        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(inputs)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)

        shortcut = MaxPooling1D(pool_size=1,
                                strides=1)(layer)

        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(layer)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Dropout(config.drop_rate)(layer)
        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(layer)
        layer = add([shortcut, layer])
        return layer

    def main_loop_blocks(layer, config):
        filter_length = config.filter_length
        n_blocks = 2
        for block_index in range(n_blocks):
            def zeropad(x):
                """
                zeropad and zeropad_output_shapes are from
                https://github.com/awni/ecg/blob/master/ecg/network.py
                """
                y = K.zeros_like(x)
                return K.concatenate([x, y], axis=2)
```

```
            def zeropad_output_shape(input_shape):
                shape = list(input_shape)
                assert len(shape) == 2
                shape[2] *= 2
                return tuple(shape)

            subsample_length = 16 if block_index % 2 == 0 else 8
            shortcut = MaxPooling1D(pool_size=subsample_length)(layer)

            if block_index % 4 == 0 and block_index > 0:
                shortcut = Lambda(zeropad,
output_shape=zeropad_output_shape)(shortcut)
                filter_length *= 2

            layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Conv1D(filters=filter_length,
                           kernel_size=config.kernel_size,
                           padding='same',
                           strides=subsample_length,
                           kernel_initializer='he_normal')(layer)
            layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(config.drop_rate)(layer)
            layer = Conv1D(filters=filter_length,
                           kernel_size=config.kernel_size,
                           padding='same',
                           strides=1,
                           kernel_initializer='he_normal')(layer)
            layer = add([shortcut, layer])
        return layer

    def output_block(layer, config):
        filter_length = config.filter_length
        layer = Conv1D(filters=filter_length,
                       kernel_size=16,
                       padding='same',
                       strides=4,
                       kernel_initializer='he_normal')(layer)
        from keras.layers.wrappers import TimeDistributed
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        outputs = TimeDistributed(Dense(len_classes,
activation='softmax'))(layer)
        model = Model(inputs=inputs, outputs=outputs)

        adam = Adam(lr=0.5, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
        model.compile(optimizer=adam,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        model.summary()
        return model

    classes = ['n', 'a', 'p', 'r', 'l']
    len_classes = len(classes)
```

```
    inputs = Input(shape=(config.input_size, 1), name='input')
    layer = first_conv_block(inputs, config)
    layer = main_loop_blocks(layer, config)
    return output_block(layer, config)
```

C. Code for 19-layer Network

```
from __future__ import division, print_function

from keras import backend as K
from keras.layers import Input, Conv1D, Dense, add, Dropout, MaxPooling1D,
Activation, BatchNormalization, \
    Lambda
from keras.models import Model
from keras.optimizers import Adam

from config import get_config


def ECG_model(config):
    """
    reference to codes at
    https://github.com/awni/ecg/blob/master/ecg/network.py
    and
    https://github.com/physhik/ecg-mit-bih/blob/master/src/graph.py
    """

    def first_conv_block(inputs, config):
        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(inputs)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)

        shortcut = MaxPooling1D(pool_size=1, strides=1)(layer)

        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(layer)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Dropout(config.drop_rate)(layer)
        layer = Conv1D(filters=config.filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(layer)
        layer = add([shortcut, layer])
        return layer
```

```
def main_loop_blocks(layer, config):
    filter_length = config.filter_length
    n_blocks = 7
    for block_index in range(n_blocks):
        def zeropad(x):
            """
            zeropad and zeropad_output_shapes are from
            https://github.com/awni/ecg/blob/master/ecg/network.py
            """
            y = K.zeros_like(x)
            return K.concatenate([x, y], axis=2)

        def zeropad_output_shape(input_shape):
            shape = list(input_shape)
            assert len(shape) == 3
            shape[2] *= 2
            return tuple(shape)

        subsample_length = 4 if block_index % 2 == 0 else 1
        shortcut = MaxPooling1D(pool_size=subsample_length)(layer)
        if block_index % 4 == 0 and block_index > 0:
            shortcut = Lambda(zeropad,
output_shape=zeropad_output_shape)(shortcut)
            filter_length *= 2

        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Conv1D(filters=filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=subsample_length,
                       kernel_initializer='he_normal')(layer)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        layer = Dropout(config.drop_rate)(layer)
        layer = Conv1D(filters=filter_length,
                       kernel_size=config.kernel_size,
                       padding='same',
                       strides=1,
                       kernel_initializer='he_normal')(layer)
        layer = add([shortcut, layer])
    return layer

def output_block(layer, config):
    filter_length = config.filter_length
    layer = Conv1D(filters=filter_length,
                   kernel_size=16,
                   padding='same',
                   strides=2,
                   kernel_initializer='he_normal')(layer)
    from keras.layers.wrappers import TimeDistributed
    layer = BatchNormalization()(layer)
    layer = Activation('relu')(layer)
    outputs = TimeDistributed(Dense(len_classes,
activation='softmax'))(layer)
    model = Model(inputs=inputs, outputs=outputs)
```

```
        adam = Adam(lr=0.5, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
        model.compile(optimizer=adam,
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        model.summary()
        return model

    classes = ['n', 'a', 'p', 'r', 'l']
    len_classes = len(classes)

    inputs = Input(shape=(config.input_size, 1), name='input')
    layer = first_conv_block(inputs, config)
    layer = main_loop_blocks(layer, config)
    return output_block(layer, config)
```

C. Code for 35-layer network

```
from __future__ import division, print_function

from keras.models import Model
from keras.layers import Input, Conv1D, Dense, add, Flatten,
Dropout,MaxPooling1D, Activation, BatchNormalization, Lambda,
GlobalAveragePooling2D
from keras import backend as K
from keras.optimizers import Adam
from keras.layers.wrappers import TimeDistributed

from config import get_config


def ECG_model(config):
     """
    reference to codes at
    https://github.com/awni/ecg/blob/master/ecg/network.py
    and
    https://github.com/physhik/ecg-mit-bih/blob/master/src/graph.py
    """
    def first_conv_block(inputs, config):
        layer = Conv1D(filters=config.filter_length,
               kernel_size=config.kernel_size,
               padding='same',
               strides=1,
               kernel_initializer='he_normal')(inputs)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)

        shortcut = MaxPooling1D(pool_size=1,
                      strides=1)(layer)

        layer =  Conv1D(filters=config.filter_length,
               kernel_size=config.kernel_size,
               padding='same',
               strides=1,
               kernel_initializer='he_normal')(layer)
        layer = BatchNormalization()(layer)
```

```
        layer = Activation('relu')(layer)
        layer = Dropout(config.drop_rate)(layer)
        layer =  Conv1D(filters=config.filter_length,
                        kernel_size=config.kernel_size,
                        padding='same',
                        strides=1,
                        kernel_initializer='he_normal')(layer)
        layer = add([shortcut, layer])
        return layer

    def main_loop_blocks(layer, config):
        filter_length = config.filter_length
        n_blocks = 15
        for block_index in range(n_blocks):
            def zeropad(x):
                """
                zeropad and zeropad_output_shapes are from
                https://github.com/awni/ecg/blob/master/ecg/network.py
                """
                y = K.zeros_like(x)
                return K.concatenate([x, y], axis=2)

            def zeropad_output_shape(input_shape):
                shape = list(input_shape)
                assert len(shape) == 3
                shape[2] *= 2
                return tuple(shape)

            subsample_length = 2 if block_index % 2 == 0 else 1
            shortcut = MaxPooling1D(pool_size=subsample_length)(layer)

            if block_index % 4 == 0 and block_index > 0 :
                shortcut = Lambda(zeropad,
output_shape=zeropad_output_shape)(shortcut)
                filter_length *= 2

            layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer =  Conv1D(filters= filter_length,
                            kernel_size=config.kernel_size,
                            padding='same',
                            strides=subsample_length,
                            kernel_initializer='he_normal')(layer)
            layer = BatchNormalization()(layer)
            layer = Activation('relu')(layer)
            layer = Dropout(config.drop_rate)(layer)
            layer =  Conv1D(filters= filter_length,
                            kernel_size=config.kernel_size,
                            padding='same',
                            strides= 1,
                            kernel_initializer='he_normal')(layer)
            layer = add([shortcut, layer])
        return layer

    def output_block(layer, config):
        filter_length = config.filter_length
        layer = Conv1D(filters=filter_length,
```

```
                        kernel_size=16,
                        padding='same',
                        strides=2,
                        kernel_initializer='he_normal')(layer)
        layer = BatchNormalization()(layer)
        layer = Activation('relu')(layer)
        outputs = TimeDistributed(Dense(len_classes,
activation='softmax'))(layer)
        model = Model(inputs=inputs, outputs=outputs)

        adam = Adam(lr=0.5, beta_1=0.9, beta_2=0.999, epsilon=None,
decay=0.0, amsgrad=False)
        model.compile(optimizer= adam,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
        model.summary()
        return model

    classes = ['n','a','p','r','l']
    len_classes = len(classes)

    inputs = Input(shape=(config.input_size, 1), name='input')
    layer = first_conv_block(inputs, config)
    layer = main_loop_blocks(layer, config)
    return output_block(layer, config)
```

D. Code for Data Preprocessing

```
def dataprocess(nclasses):
    import scipy.io as scio
    import numpy as np
    from sklearn.model_selection import train_test_split

    #load data
    def loaddata(label_class, datafile):
        data = scio.loadmat(datafile)
        _, _, _,  key = data.keys()
        data = data[key]
        data = data[14:526, :]
        m, n = np.shape(data)
        label = np.zeros(shape=[n, nclasses], dtype=float)

        label[:, label_class] = 1
        data = np.transpose(data)
        return data, label

    #concate data
    n_data, n_label = loaddata(0, 'dataset/normalrr1.mat')
    x, y = n_data, n_label
    a_data, a_label = loaddata(1, 'dataset/apcrr2.mat')
    x = np.concatenate((x, a_data))
    y = np.concatenate((y, a_label))
    p_data, p_label = loaddata(2, 'dataset/pvcrr2.mat')
    x = np.concatenate((x, p_data))
    y = np.concatenate((y, p_label))
    r_data, r_label = loaddata(3, 'dataset/rbbbrr1.mat')
```

```
    x = np.concatenate((x, r_data))
    y = np.concatenate((y, r_label))
    l_data, l_label = loaddata(4, 'dataset/lbbbrr1.mat')
    x = np.concatenate((x, l_data))
    y = np.concatenate((y, l_label))

    #produce 20 trainingset and a test set
    train_x, train_y = dict(), dict()
    x_val, y_val = list(), list()
    x, x_val, y, y_val = train_test_split(x, y, test_size=0.1,
random_state=None)
    for i in range(20):
        train_x[i], train_y[i] = list(), list()
        train_x[i], x_test, train_y[i], y_test = train_test_split(x, y,
test_size=0.1, random_state=None)
    print('datatype{},datashape{}'.format(type(a_data), np.shape(a_data)))
    print('labeltype{},labelshape{}'.format(type(a_label),
np.shape(a_label)))
    return train_x, train_y, x_val, y_val


if __name__ == '__main__':
    dataprocess(5)
```

E. Code for Training and Testing

```
from __future__ import division, print_function

import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
from keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard,
ReduceLROnPlateau, LearningRateScheduler
from sklearn.model_selection import train_test_split

from config import get_config
from data import dataprocess

def train(config, xtr, ytr, xte, yte, model_number):
    if model_number == 9:
        from graph_9 import ECG_model
    elif model_number == 19:
        from graph_19 import ECG_model
    else:
        from graph_34 import ECG_model

    classes = ['n','a','p','r','l']
    print('X', np.shape(xtr))
    print('y', np.shape(ytr))
    print('X type', type(xtr))
    print('y type', type(ytr))
    xtr = np.expand_dims(xtr, axis=2) # reshape data according to Keras'
requirements
    xte = np.expand_dims(xte, axis=2)
    (m, n) = ytr.shape
```

```
    ytr = ytr.reshape((m, 1, n))
    (mvl, nvl) = yte.shape
    yte = yte.reshape((mvl, 1, nvl))
    print('xte', np.shape(xte))
    print('yte', np.shape(yte))

    model = ECG_model(config)
    initial_epoch = 0
    mkdir_recursive('models')
    callbacks = [
            EarlyStopping(patience = config.patience, verbose=1),
            ReduceLROnPlateau(factor = 0.2, patience = 3, min_lr = 0.01,
verbose=1),
            TensorBoard( log_dir='./logs', histogram_freq=0, write_graph =
True, write_grads=False, write_images=True),
            ModelCheckpoint('models/{}-latest.hdf5'.format(config.feature),
monitor='val_loss', save_best_only=False, verbose=2, period=10)
    ]

    model.fit(xtr, ytr,
              validation_data=(xte, yte),
              epochs=config.epochs,
              batch_size=config.batch,
              callbacks=callbacks,
              initial_epoch=initial_epoch)



def main(config):
    print('feature:', config.feature)
    print('drop rate:', config.drop_rate)
    print('split', config.split)
    print('input_size', config.input_size)
    for i in range(20):
        xtr, ytr, xte, yte = dataprocess(5)
        print('training set', i)
        train(config, xtr[i], ytr[i], xte, yte, 9)
        train(config, xtr[i], ytr[i], xte, yte, 19)
        train(config, xtr[i], ytr[i], xte, yte, 34)


if __name__=="__main__":
    config = get_config()
    main(config)
```

Reference

[1] M. Llamedo and J.P. Martinez, "Heartbeat classification using feature selection driven by database generalization criteria," *IEEE Trans. Biomed. Eng.*, vol. 58, pp.616–625, 2011.

[2] P. De Chazal, M. O'Dwyer and R. B. Reilly, "Automatic classification of heartbeats using ECG morphology and heartbeat interval features," *IEEE Trans. Biomed.*, vol.51, pp.1196–1206, 2004.

[3] Y. Kutlu and D. Kuntalp, "Feature extraction for ECG heartbeats using higher order statistics of WPD coecients," *Comput. Methods Progr. Biomed.*, vol. 105, pp.257–267, 2012.

[4] P. De Chazal and R. B. Reilly, "A patient-adapting heartbeat classier using ECG morphology and heartbeat interval features," *IEEE Trans. Biomed. Eng.*, vol. 53, pp.2535–2543, 2006.

[5] G. de Lannoy, D. Francois, J. Delbeke and M. Verleysen," Weighted conditional random fields for supervised interpatient heartbeat classification," *IEEE Trans. Biomed.*, vol. 59 pp.241–247, 2011.

[6] P.-C. Chang, J.-J. Lin, J.-C. Hsieh and J. Weng, "Myocardial infarction classification with multi-lead ECG using hidden Markov models and Gaussian mixture models," *Appl. Soft Comput.*, vol. 12, pp. 3165–3175, 2012.

[7] X. Lei, Y. Zhang, and Z. Lu, "Deep learning feature representation for electrocardiogram identification," *Proc. 2016 IEEE International Conference on Digital Signal Processing (DSP)*, pp. 11-14., 2016.

[8] P. Rajpurkar, A. Y. Hannun, M. Haghpanahi, C. Bourn, and A. Y. Ng, "Cardiologist level arrhythmia detection with convolutional neural networks," arXiv: 1707.01836v1, July 2017.

[9] L. S. Lilly, "Pathophysiology of heart disease: A collaborative project of medical students and faculty," Retrieved from http://ebookcentral.proquest.com, 2015.

[10] "Electrocardiography" Wikipedia: The Free Encyclopedia. 4 March 2020, 01:35 (UTC) from http://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350.

[11] C. C. Aggarwal, *Neural Networks and Deep Learning – A Textbook*, Springer, 2018.

[12] W. Pitts and W. McCulloch, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, pp. 115-133, 1943.

[13] H. J. Kelley, “Gradient theory of optimal flight paths,” *ARS Journal*, vol. 30, no.10, pp. 947–954, 1960.

[14] S. Dreyfus, “The numerical solution of variational problems,” *Journal of Mathematical Analysis and Applications.*, vol. 5, no. 1, pp. 30-45, 1962.

[15] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks.*, vol. 61, pp. 85–117, 2015.

[16] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[17] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation.*, vol. 9 , no. 8, pp. 1735–1780, 1997.

[18] D. Reinsel, J. Gantz, and J. Rydning, "The digitization of the world from edge to core," IDC White Paper, November 2018.

[19] A. Krizhevsky, I. Sutskever and G. Hinton, “ImageNet classification with deep convolutional neural networks,” *NIPS Conference*, pp. 1097–1105, 2012.

[20] S. Gao, M. Cheng, K. Zhao, X. Zhang, M. Yang and P. H. S. Torr, "Res2Net: A new multi-scale backbone architecture," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p.1, 2019.

[21] J. Chen, *Deep Learning for Handwritten Digits Recognition Using MATLAB Toolbox*, Project Report, University of Victoria, 2019.

[22] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proc. IEEE*, vol. 86, pp. 2278–2324, Nov. 1998.

[23] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” arXiv:1502.03167, 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” *Proc. IEEE Conf. Computer Vision and pattern Recognition*, pp. 770-778, 2016.

[25] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[26] G. B. Moody and R. G. Mark, “The impact of the MIT-BIH Arrhythmia Database,” *IEEE Eng in Med and Biol.*, vol. 20, no. 3, pp. 45-50, May-June 2001.

[27] J. Pan and W. J. Tompkins, “A Real-Time QRS Detection Algorithm,” *IEEE Transactions on Biomedical Engineering*, vol. BME-32, no. 3, pp. 230-236, March 1985.

[28] W. Yang, Y. Si, D. Wang and B. Guo, “Automatic recognition of arrhythmia based on principal component analysis network and linear support vector machine,” *Comput. Biol. Med.*, vol.101, pp. 22–32, 2018.

[29] U. R. Acharya, S. L. Oh, Y. Hagiwara, J. H. Tan, M. Adam, A. Gertych and R. S. Tan, “A deep convolutional neural network model to classify heartbeats,” *Comput. Biol. Med.*, vol. 89, pp. 389–396, 2017.

[30] S. L. Oh, E. Y. Ng, R. S. Tan and U. R. Acharya, “Automated diagnosis of arrhythmia using combination of CNN and LSTM techniques with variable length heart beats,” *Comput. Biol. Med.*, vol. 102, pp. 278-287, 2018.

[31] O. Yildirim, “A novel wavelet sequences based on deep bidirectional LSTM network model for ECG signal classification,” *Comput. Biol. Med.,* vol. 96, pp. 189–202, 2018.

[32] O. Yıldırıma, P. Pławiakb, R. S. Tan, and U. R. Acharya, “Arrhythmia detection using deep convolutional neural network with long duration ECG signals,” *Comput. Biol. Med.*, vol. 102, pp. 411-420, 2018.