Convolutional Neural Network Integration to a 3-D Ray-traced Biological Neural Network

by

Xuan Chen
B.Eng., Shanghai University of Electric Power, 2021

A Report Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Xuan Chen, 2024
University of Victoria

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

Convolutional Neural Network Integration to a 3-D Ray-traced Biological Neural Network

by

Xuan Chen
B.Eng., Shanghai University of Electric Power, 2021

Supervisory Committee

Dr. Tao Lu, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Aaron Gulliver, Departmental Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Tao Lu, Supervisor
(Department of Electrical and Computer Engineering)

Dr. Aaron Gulliver, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

This study presents a machine learning model that integrates a Convolutional Neural Network (CNN) into a 3-D Ray-traced Biological Neural Network (RayBNN). RayBNN specializes in rearranging and adapting to various problems through transfer learning. CNN is renowned for extracting features from data efficiently, which may boost to RayBNN performance. In this report, two integration schemes are implemented and tested on the Modified National Institute of Standards and Technology(MNIST) and Wakefulness Test recordings (MWT) datasets respectively. Using MNIST dataset, we trained a CNN with an artificial neural network (ANN) and an auto encoder/decoder to extract features from datasets and used them as the input of RayBNN. Using the CNN with ANN approach, an accuracy of 0.9919 ± 0.0012, a precision of 0.9921 ± 0.0008, a recall of 0.9922 ± 0.0012 and an F1-score of 0.9920 ± 0.0008 were obtained. When using CNN with auto encoder/decoder for feature extraction, the accuracy, precision, recall, and 0.9905 ± 0.0035, 0.9901 ± 0.0050, 0.9881 ± 0.0068, and F1-score at 0.9908 ± 0.0010 respectively. For MWT dataset, Cohen's Kappa values of 0.68 ± 0.05, 0.71 ± 0.04, 0.04 ± 0.02, and 0.06 ± 0.02 for Wakefulness, Microsleep Episode, Microsleep Episode Candidate, Episodes of Drowsiness classes were obtained using CNN with ANN to extract features for RayBNN.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables vii

List of Figures viii

Acknowledgements x

1 Introduction 1

1.1 Outline 3

1.2 Contribution 3

2 Background 4

2.1 Biological Neural Network 4

2.2 Neural Networks in Machine Learning 7

2.2.1 Feed-Forward Neural Network (FNN) 9

2.2.2 Convolutional Neural Network (CNN) 10

2.2.3 Autoencoder 14

2.2.4 Transfer Learning and RayBNN 16

2.3 Overfitting and Underfitting 21

2.4 Evaluation metrics 22

3 Datasets and Methodology 25

3.1 Datasets 25

3.1.1 MNIST dataset 25

3.1.2 MWT dataset 26

3.2 Methodology . . . 30
3.2.1 CNN+ANN approach with RayBNN . . . 30
3.2.2 CNN+Autoencoder approach with RayBNN . . . 31
3.3 Preparation for RayBNN API . . . 33
3.4 Hardware . . . 33
3.5 Framework . . . 34

4 MNIST dataset **36**
4.1 Results . . . 37
4.1.1 Feature representations of the models . . . 37
4.1.2 Comparisons and analysis . . . 41

5 MWT dataset **46**
5.1 Hyper-parameter tuning . . . 46
5.2 Results . . . 48

6 Conclusions **55**

Bibliography **57**

List of Tables

Table 3.1 Dataset Distribution for training, validation, and testing. 29
Table 3.2 Cluster Specifications. 34

List of Figures

Figure 2.1 Confocal image of rat neurons and glial grown in culture and stained with antibodies to microtubule [35]. 4
Figure 2.2 Schematic of a Biological Neuron [7]. 5
Figure 2.3 Schematic of a Biological Neural Network [7]. 6
Figure 2.4 The first neural network created by Warren MuCulloch and Walter Pitts in 1943 [21]. 8
Figure 2.5 An example of the feed-forward neural network. 9
Figure 2.6 Overview of a CNN architecture. 10
Figure 2.7 Image of Abraham Lincoln being processed as matrix pixels [16]. 10
Figure 2.8 CNN convolution process. 11
Figure 2.9 The feature representation of Lincoln's image after convolutional layer. 12
Figure 2.10 Scaling down using Max Pooling. 13
Figure 2.11 The feature representation of Lincoln's image after max-pooling layer. 14
Figure 2.12 Autoencoder structure. 15
Figure 2.13 The original and reconstructed images from an autoencoder. . . 16
Figure 2.14 Stimulated 3d ray traced biological neural network (RayBNN) [42]. 18
Figure 2.15 *(a) Location of input neurons at the surface of the network sphere. (c) Transfer of input neurons to a new network sphere where the dimension of the data is densified. (b) If the new dataset concatenates the old dataset, then the old neurons migrate to the north while new neurons are created in the south of the new network sphere. [42].* 19
Figure 2.16 Overview Algorithm for RayBNN [42]. 21

Figure 3.1 Image examples in MNIST dataset [8]. 26

Figure 3.2 Electrode locations of International 10-20 system for EEG (electroencephalography) recording [25]. 28
Figure 3.3 CNNs+ANN, and applied to RayBNN. 30
Figure 3.4 CNN Autoencoder+RayBNN model structure. 32
Figure 3.5 Transposed convolution working process. 33

Figure 4.1 Feature representation in the first convolutional layer. 37
Figure 4.2 Feature representation in the first max-pooling layer. 38
Figure 4.3 Feature representation in the second convolutional layer. 39
Figure 4.4 Feature representation in the second max-pooling layer. 39
Figure 4.5 Reconstructed output of CNN autoencoder model. 40
Figure 4.6 Accuracy of four models at different epochs. 41
Figure 4.7 Precision of four models at different epochs. 42
Figure 4.8 Recall of four models at different epochs. 43
Figure 4.9 F1-score of four models at different epochs. 44

Figure 5.1 Cohen's Kappa of CNN+ANN with different repetitions at 2nd blocks. 47
Figure 5.2 Cohen's Kappa of CNN+ANN with different repetitions at 3rd blocks. 47
Figure 5.3 Kappa values of CNN+ANN and CNN+ANN with RayBNN models for W classification at different epochs. The yellow line indicates the Cohen's Kappa value from experts. 50
Figure 5.4 Kappa values of CNN+ANN and CNN+ANN with RayBNN models for MSE classification at different epochs. The yellow line indicates the Cohen's Kappa value from experts. 51
Figure 5.5 Kappa values of CNN+ANN and CNN+RayBNN models for MSEc classification at different epochs.The yellow line indicates the Cohen's Kappa value from experts. 52
Figure 5.6 Kappa values of CNN+ANN and CNN+RayBNN models for ED classification at different epochs.The yellow line indicates the Cohen's Kappa value from experts. 53

ACKNOWLEDGEMENTS

I would like to thank:

Jasper and Kylie, for supporting me in the low moments.

Supervisor Dr.Tao Lu, for mentoring, support, encouragement, and patience.

Finally, I would like to extend my appreciation to my families and friends for their unwavering support and understanding throughout this journey.

Xuan Chen

Chapter 1

Introduction

Neural networks are essentially computational models that are typically configured for distinct functions and tailored to process specific datasets. When facing different problems, changing the dimension or format of data is needed. For example, when using a Convolutional Neural Network (CNN) to efficiently identify objects in 1024×1024-pixel high-resolution RGB images, one may start training on the down-sampled 256×256 low-resolution images first before training the full-resolution images. In addition, one may also train grayscale images first before training the full color RGB images. Both approaches require the change of data dimensions or format to the CNN which sometimes requires reconstructing and retraining the entire neural network. Moreover, models that are adept at classification tasks, might not exhibit the same level of proficiency when applied to disparate tasks, such as reinforcement learning or quantification, revealing an inherent inflexibility in their design. This underscores the need for more versatile neural network architectures that can maintain performance across a broader spectrum of tasks and data variations.

Transfer learning [1, 18, 23, 24, 34] is specialized in addressing such limitations. It often involves taking a neural network that has already been trained on a dataset and fine-tuning it for another related and distinct one. This approach can substantially reduce the time needed for training and even makes it possible to train the model on a less robust computer. However, to maximize learned knowledge and adapt the model to perform optimally under new conditions, the transferred models need to match the dimensions of the original model and modify the neural architecture to analyze new datasets. Current transfer learning is highly constrained by its structure. Meanwhile, it isn't always as effective as we would want. For example, a CNN model (e.g., ResNet-50 [13]) is initially trained on a large image dataset (ImageNet [31]) catego-

rized into thousands of classes, which has images resized to a uniform dimension of 244×244 pixels. If one wants to use this pre-trained CNN model to classify 512×512-pixel satellite images to different land usage catergories, the satellite images must be resized to 244×244 pixels before being fed into the ResNet-50, which inevitably causes a loss of detailed information. Besides, land usage categories are fundamentally different from classes defined in ImageNet, which requires the new output layer to be trained from scratch. These limitations should all be considered critically because they restrict the practical applicability of neural networks in real-world applications.

The new transfer learning model, which is known as 3-D ray-traced biological neural network(RayBNN) [42], is created by mimicking the biological neural network(BNN) to address the limitations of transfer learning. Ray tracing is a method of calculating particle or wave paths that can be used to simulate the way biological neurons transmit signals through complex media, focusing on the spatial relationships and interactions of neurons. Instead of the traditional well-defined layers, the neurons in RayBNN are connected randomly to enable information transfers more efficiently. RayBNN enables the network to grow into different shapes and sizes, so that it can handle a large number of datasets and be applied to any other network designs. On the other hand, the complexity of the RayBNN is highly dependent on input data dimensions. Therefore, extracting essential features from data through a highly efficient data preprocessing method may significantly simplify RayBNN structure and enhance its performance.

In data sciences, CNN stands out as exceptionally proficient at feature extraction, due to its design and functionality. Layers in CNNs are designed to perform specific transformations of the input data for abstracting features at various levels. For example, the convolutional layer can extract feature information whenever it occurs, such as edges, colors, by applying filters to detect types of features in the input and sliding over the entire input image. Besides, CNN is set to learn hierarchical representation using its multi-layers where initial layers detect simple features, and deeper layers compile these features into more complex representations. This kind of hierarchical feature learning allows CNN to understand the context and detail of the data.

For these uniques properties, CNN was used as a pivotal tool of feature extraction in this project. Coupled with RayBNN, the CNN effectively preprocesses the data, ensuring that the RayBNN receives the most pertinent information, thereby streamlining the subsequent learning process. By leveraging the RayBNN's flexibility of dealing with different types and formats of datasets with CNNs, this integration

model is developed as a powerful learning and adaptation tool that can solve the limitations of traditional transfer learning model, especially in tasks that require a high degree of adaptability and cognitive functionality.

1.1 Outline

This report is outlined as follows:

Chapter 1 introduces the project and describes the problems being solved.

Chapter 2 provides the background of the research.

Chapter 3 details CNN and RayBNN integration shemes.

Chapter 4 verifies the integration on MNIST dataset.

Chapter 5 applies the models to MWT dataset, and analyzes the results.

Chapter 6 concludes the project report with suggestions for future research.

1.2 Contribution

In my MEng project, I undertook the integration of CNN with RayBNN, which enhanced the efficiency for real-world applications. I further characterized its performance and analyzed the results while RayBNN API was designed and implemented by Brosnan Yuen.

Chapter 2

Background

2.1 Biological Neural Network

The brain is the core control unit of the neural network of living beings, having specialized subunits in charge of vision, sensing, movement, and hearing. This sophisticated organ is linked to the body's sensors and actors by a thick network of nerves, allowing for continuous interaction and cooperation. A biological neural network in the brain as shown in Fig. 2.1 is like a big network of tiny processing units called neurons.

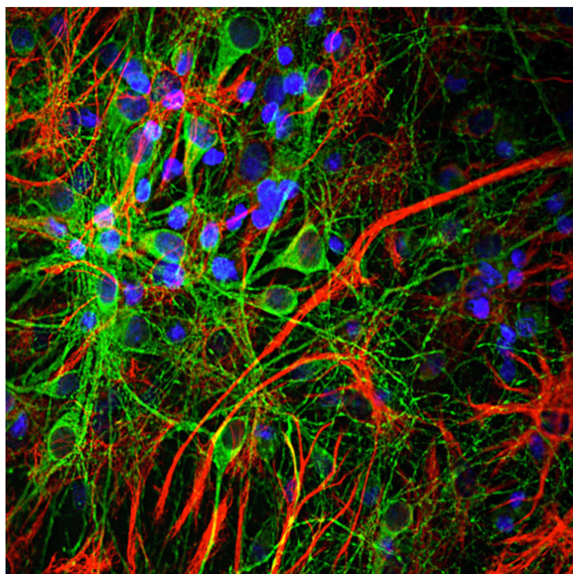


Figure 2.1: Confocal image of rat neurons and glial grown in culture and stained with antibodies to microtubule [35].

The brain is made up of around 10^{11} neurons, which form the foundation of the

central nervous system in a live organism. These neurons collaborate to process information, make decisions, and carry out actions, making them critical to the operation of every living body. Each neuron has its own level of activity or activation, which depends on the inputs it gets. The neuron is the basic building block of neural networks. A neuron, like all other bodily cells, has a DNA code and goes through the same creation process. Despite potential DNA changes, the neuron's function is const the dendrites, and the axon as shown in Fig. 2.2. Dendrites resemble fibers that branch out in many directions and link with several cells within the same cluster. [7]

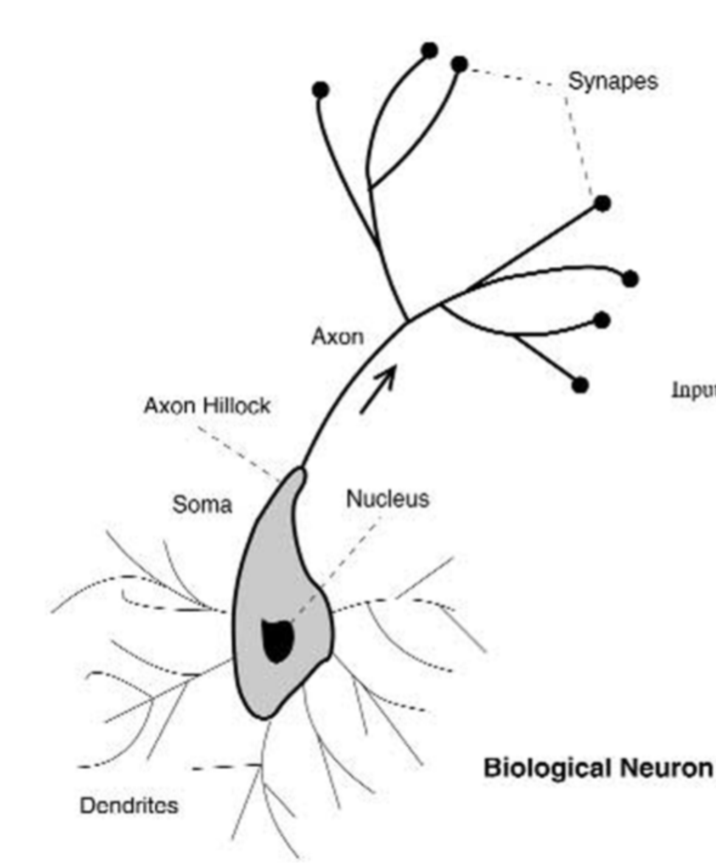


Figure 2.2: Schematic of a Biological Neuron [7].

Generally, neurons are specialized nerve cells that contact the activity with one another neuron using electrical and chemical signals. Even though a neuron can only send one signal at a time, this signal can reach multiple neurons. Dendrites, or branching extensions of the neuron, receive information from nearby neurons. These impulses are subsequently carried via the neuron's cell body and along a long, thin fiber known as the axon. The axon transports electrical impulses out from the cell

body and to neighboring neurons or target cells. The signal travels up the axon until it reaches the axon terminal, where it contacts the dendrites of another neuron via a connection known as a synapse. This synaptic connection permits the signal to be sent from one neuron to the next, resulting in a chain reaction or domino effect that facilitates sophisticated communication within the system.

Each neuron usually has just one axon, which can vary in length and is responsible for delivering the neuron's output signal to neighboring cells [41]. So in a biological network as shown in Fig. 2.3, imagine dendrites as little messengers in our brain's neurons. They pick up signals from other neurons and pass them along to the soma. But the soma, along with its nucleus, does not do much with the messages it receives. It is more focused on keeping the neuron healthy and functioning. While it is not directly involved in processing the incoming and outgoing data, soma is important for making sure everything runs smoothly [2].

Figure 2.3: Schematic of a Biological Neural Network [7].

2.2 Neural Networks in Machine Learning

According to Ref. [21], neural connections can be depicted by sentential logic, which is the means of combining or substituting propositions to define more complex statements. Meanwhile, to any logical expression that satisfies specific conditions, we can always find a net to describe it. The neural networks contain lots of small units called "neurons" that can process information based on a specific logical regulation. Between neurons, they are set to connect and transmit signals. In this way can a network compute logical or arithmetic functions. In the same article, it was also demonstrated that a network of artificial neurons can mimic a digital computer. The proposed model is simple and clear as shown in Fig. 2.4. It only contains 2 parts: the first part-g takes the inputs and calculates the aggregation value. The second part-f is defined for decision-making. This model provides an understandable framework of a computation based on a network, which builds a foundation for the development of neural networks later. This model also has its limitations. For example, it does not account for the dynamic and adaptive nature of biological neural networks, such as learning and memory. It assumes that the connections between neurons are fixed, without considering the synaptic plasticity in biological systems.

Despite these limitations, this model has a significant impact on the fields of artificial intelligence and computational neuroscience. It displays how neural networks can be used to simulate intelligent behavior and perform complex computations, which means that it is possible to consider the logic of computation in human brains as a solution to computing networks.

Ref. [21] also inspires subsequent research in perceptron [29], which refers to a type of artificial neuron. It consists input units that represent sensory neurons, weights that simulate synaptic strengths, bias that adjust the threshold, and an output unit that functions similarly to a motor neuron. The perceptron accepts input signals, multiplies them by their weights, and then sums the results [28]. After a bias added to the input sum to help perceptron learn more complex patterns, and adjust the threshold at which the activation function triggers, the perceptron uses an activation function, which is a step function originally, to decide whether to output a 1 or a 0 based on the weighted sum of its inputs. This decision-making process, where a simple non-linear function can represent a boundary or decision threshold, parallels how individual neurons in the brain may activate based on the collective inputs they receive from other neurons. The perceptron model also showed that information is stored in

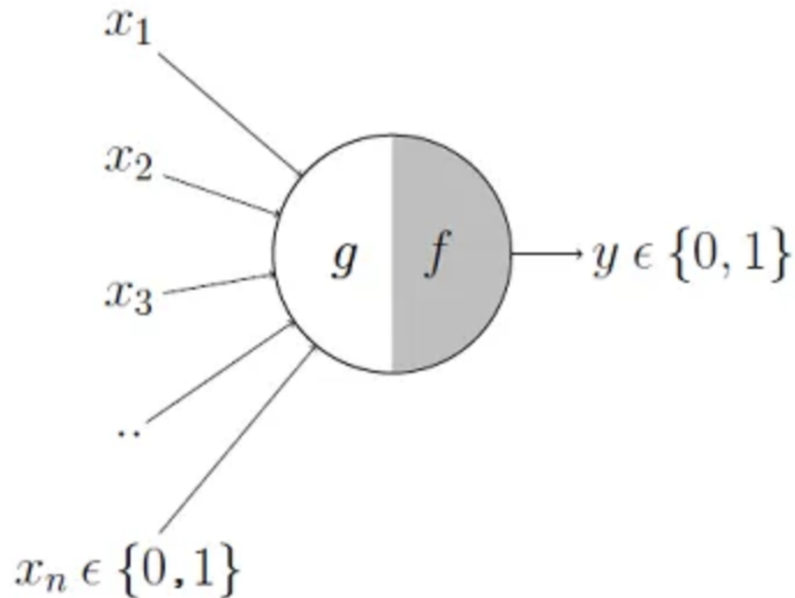


Figure 2.4: The first neural network created by Warren MuCulloch and Walter Pitts in 1943 [21].

the weights of the connections between input features and the neurons. These weights adjust as the perceptron learns from the training data. Besides, it processed the information in a dispersed way, which resembles the distributed synaptic strengths in the brain, where memories and knowledge are not stored in single neurons but rather in the patterns of connections and activations across many neurons. However, we can also see some limitations in the study of perceptrons. They can only learn linearly separable patterns, which means if the data can not be separated into classes by a straight line or hyperplane in higher dimensions, the perceptron will not be able to converge on a solution [22]. This sparked more study into the capabilities and limitations of perceptons, resulting in the development of deep learning and machine learning technologies that are extensively developing today.

Ref. [12] on the perceptron paved the way for the creation of more sophisticated neural network topologies and learning algorithms, leading to Artificial Neural Network (ANN). Unlike a single-layer perceptron that can only solve linearly separable problems, ANNs are multi-layered models inspired by the human brain's structure and function. The models consist of interconnected groups of artificial neurons or nodes. Each connection can transmit a signal from one neuron to another. The re-

ceiving neuron processes the signal and then signals downstream neurons connected to it. Below are a few of ANNs that are pertinent to this project.

2.2.1 Feed-Forward Neural Network (FNN)

FNN [43, 33, 30] is one of the basic ANN architecture. As shown in Fig. 2.5, it includes an input layer, hidden layers for specific functionalities, and an output layer.

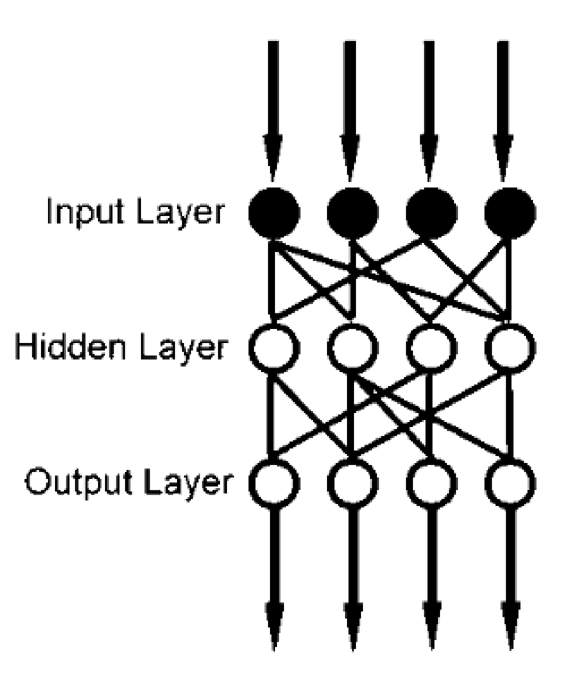


Figure 2.5: An example of the feed-forward neural network.

In hidden layers of a neural network, dense layers are frequently used as the core blocks that allow the network to learn complex patterns and relationships in the data. In dense layers, every neuron is connected to every neuron in the preceding and following layers, and are also called fully connected layers. This creates a dense web of connections, allowing them to analyze the entire set of features. The features from the previous layer are flattened into a one-dimensional vector and passed into dense layers, which compute a weighted sum of the inputs. Adding more dense layers could increase the capacity for the model to learn complex features.

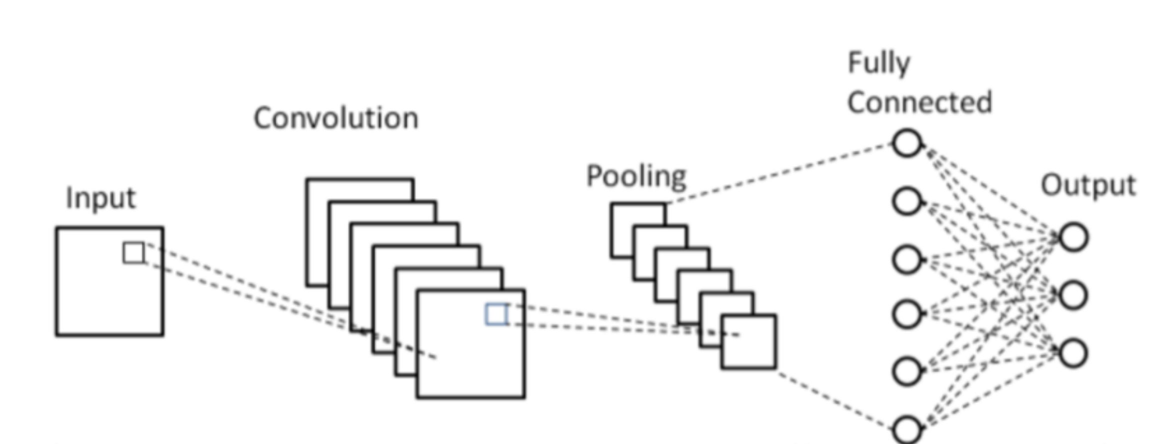
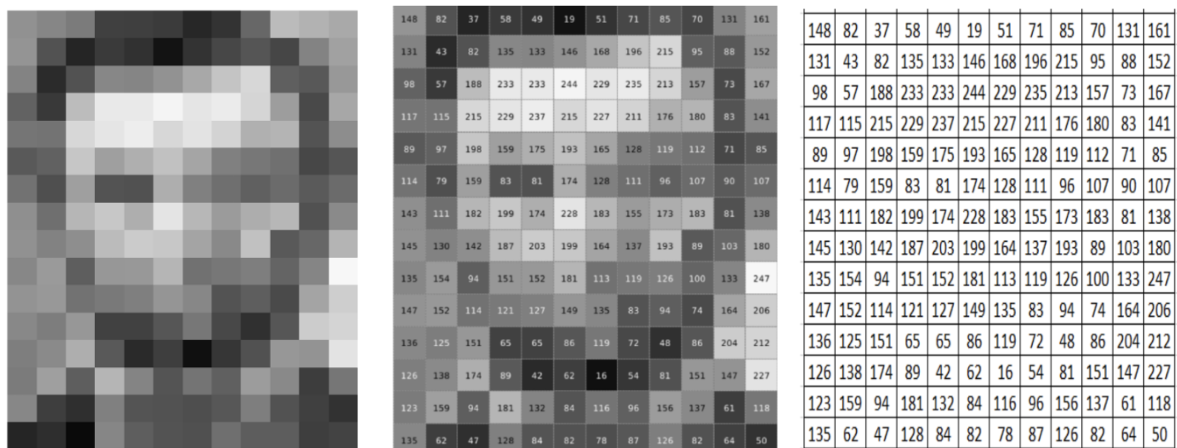


Figure 2.6: Overview of a CNN architecture.

2.2.2 Convolutional Neural Network (CNN)

CNN [39], also known as ConvNet, is a type of artificial neural network architecture particularly suited to exploit the spatial and temporal dependencies in data. CNN constructs powerful hierarchical patterns, capturing all aspects of the pre-processed data from the textures to the complex objects in an abstract understanding. Fig. 2.6 describes a common CNN structure. The networks include input, convolutional, max-pooling, fully connected, and output layers. In addition, multiple convolutional and max-pooling layers can be cascaded to form deep learning networks [3, 44].



148	82	37	58	49	19	51	71	85	70	131	161
131	43	82	135	133	146	168	196	215	95	88	152
98	57	188	233	233	244	229	235	213	157	73	167
117	115	215	229	237	215	227	211	176	180	83	141
89	97	198	159	175	193	165	128	119	112	71	85
114	79	159	83	81	174	128	111	96	107	90	107
143	111	182	199	174	228	183	155	173	183	81	138
145	130	142	187	203	199	164	137	193	89	103	180
135	154	94	151	152	181	113	119	126	100	133	247
147	152	114	121	127	149	135	83	94	74	164	206
136	125	151	65	65	86	119	72	48	86	204	212
126	138	174	89	42	62	16	54	81	151	147	227
123	159	94	181	132	84	116	96	156	137	61	118
135	62	47	128	84	82	78	87	126	82	64	50

Figure 2.7: Image of Abraham Lincoln being processed as matrix pixels [16].

To efficiently learn data patterns, like other neural networks, CNN also requires pre-processes data through normalization that would be used to convert the pixel

values to a specific range, such as 0 to 1 or -1 to 1, standardization that resizes data with same size and aspect ratio, data augmentation that provides a more diverse set for training, noise reduction that cleans up the input data to make it easier for CNN, and so on. Fig. 2.7 is an example of an image being processed as a matrix of pixels. The original image is converted to a numeric representation of the pixel values within the third sub-image. The range of values from 0 to 255 aligns with standard 8-bit grayscale image data, where 0 is black, 255 is white, and the shades of gray are represented by values in between.

The pre-processed data enters CNN for feature extraction. The functions of each layer in CNN is detailed as follows

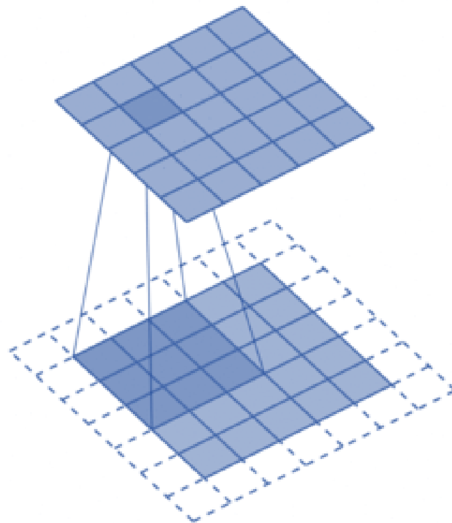


Figure 2.8: CNN convolution process.

1. **Convolutional Layer:** is designed to automatically learn spatial hierarchies of features from input data, for example, edges, corners, or other shape patterns of an image. It reduces the number of parameters in the network by reusing the same weights across the entire input. The convolutional layer uses filters or kernels, which are small grids of weights that act as feature detectors. These kernels slide across the input data (or the output of a previous layer) in specific strides. They multiply with the part of the image they are covering and sum up the result into a single output pixel in the feature. For example in Fig. 2.8, the bottom 7×7 square is the input channel of the Conv layer, the 3×3 square shade on the bottom square is the 3×3 filter, and the upper 5×5 square is the

output channel of the Conv layer. The 3×3 filter is first applied to a specific area of the blue input channel, which contains some of the picture data. The filter then performs a mathematical operation between its weights and the underlying picture pixels inside the shaded 3×3 region, which is usually a dot product.

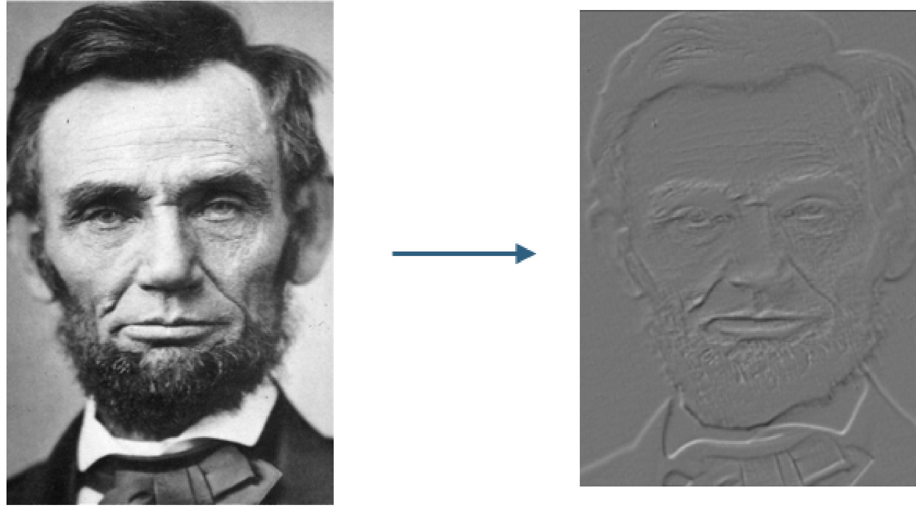


Figure 2.9: The feature representation of Lincoln's image after convolutional layer.

Fig. 2.9 demonstrate the feature representation of Lincoln's image processed by convolutional layer. The filter in convolutional layer helps in feature identification and essentially scans the whole input image by applying this operation to each new point as it slides. After that, the features to be detected are shown in terms of their intensity and existence.

2. **Max Pooling [32]:** is a downsampling technique that is frequently used in CNNs to minimize the spatial dimensions of the output from the convolutional layers. A max-pooling procedure is used to conserve the maximum value within a certain area. The dimension of the feature is significantly reduced as a result of this procedure, which effectively reduces the total number of pixels in later layers.

   For example in Fig. 2.10, in the max pooling layer, we define the number of pixels we would like the filter to move on the image is called stride, which is 2 in this example. Using the output of convolution, we determine the maximum value of the 2×2 block by starting from the first 2×2 area, which is the orange

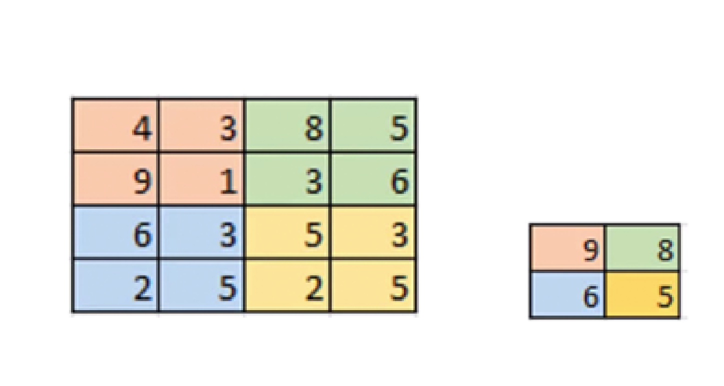


Figure 2.10: Scaling down using Max Pooling.

area of the sub-image on the left(4,3,9,1). The first number is recorded in the output channel, which is the max number "9" among 4,3,9,1. Each time we advance by the number of stride. Since we are utilizing 2, we just slide by 2 and carry out the identical action. The maximum value in the subsequent 2×2 block is determined, and stored in the output, and we continue by sliding over by 2. Mathematically, the output p_{ij} of the pooling operation at position (i, j) is given by:

$$p_{ij} = \max\left(x_{2i,2j}, x_{2i,2j+1}, x_{2i+1,2j}, x_{2i+1,2j+1}\right) \tag{2.1}$$

After sliding over the entire image (supposed the size of image $= 4 \times 4$), we will get a new 2×2 image, which is regarded as the output of the Max-pooling layer (as shown in the sub-image on the right).

Fig. 2.11 demonstrates the feature representation of Lincoln's image processed by max-pooling layer, fewer details contained in this image. The pooling operation helps to make the representation invariant to small translations and reduces the computational complexity for subsequent layers.

3. **Fully connected Layer** is usually used to combine with CNNs to learn complex relationships between features in the data. In CNN, the fully connected layer serves as a decision-making body. It integrates the learned features, which are spatially hierarchal and abstracted by the preceding layers, to form the final output predictions.

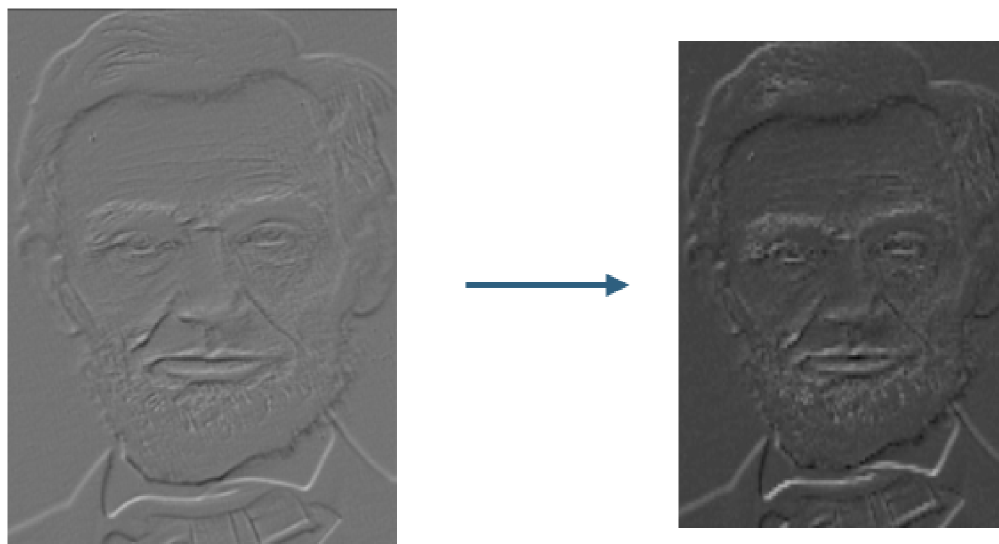


Figure 2.11: The feature representation of Lincoln's image after max-pooling layer.

2.2.3 Autoencoder

An autoencoder is one of the architectures in artificial neural networks, primarily focused on learning efficient representations of input data, often for dimensionality reduction or feature learning, consisting of two main parts: the encoder and the decoder. The encoder is the first part of the autoencoder network. It takes the input data and compresses it into a lower-dimensional representation. The layers in the encoder progressively extract more abstract and compact features from the input. The decoder is the second part of the autoencoder network. It takes the compressed representation from the encoder and attempts to reconstruct the original input data. The decoder typically mirrors the architecture of the encoder, using one or more layers to progressively reconstruct the input from the compact representation.

Fig. 2.12 presents the operational flow of an autoencoder model. The input data is initially fed into the encoder. The encoder's primary function is to compress the input into a more compact, lower-dimensional representation, facilitating the extraction and retention of essential features while reducing noise and redundancy. It serves as the preliminary feature extraction phase, preparing the data for deeper analysis.

After processed by encoder, the data is at its most compressed form, refers to the "Encoded Data" in the diagram. This stage represents the distilled essence of the input image, containing the most critical and informative features necessary for reconstructing the image or for further processing. The encoded data is a dense rep-

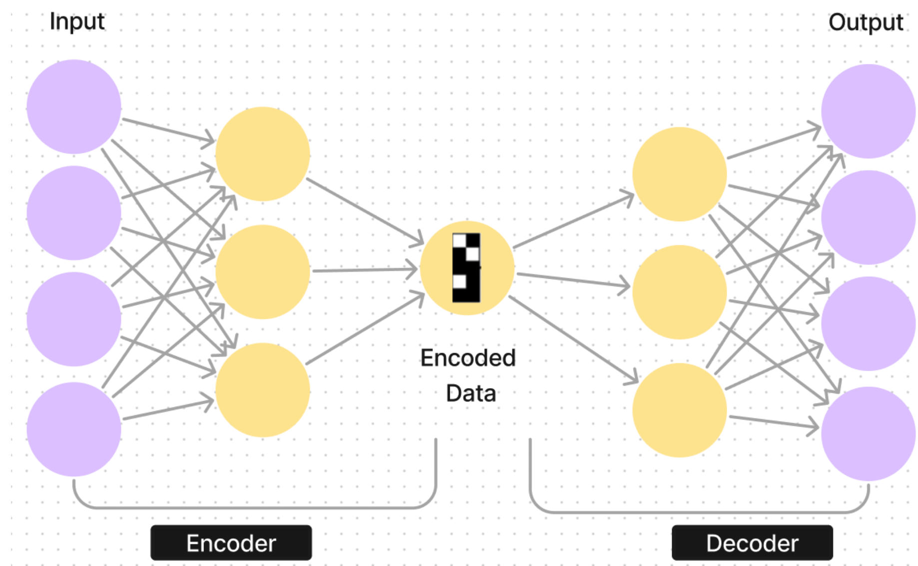


Figure 2.12: Autoencoder structure.

resentation that encapsulates the unique characteristics and patterns of the input digit, optimized for minimal information loss. The decoder mirrors the encoder's architecture but in reverse, aiming at reconstructing the original input image from the encoded data. The output of the decoder is reconstructed from the encoded data, demonstrating the autoencoder's effectiveness in preserving essential information through the encoding and decoding processes.

Fig. 2.13 displays a comparison of the original and the reconstructed costume images from an autoencoder model. The original images are a set of input data, shown with visible features, for example, the sole outline of a sneaker, a logo on a T-shirt, a clear separation of legs and visible folds, and details at the joints. The reconstructed images are the output that processed by an autoencoder. They are significantly blurred and lacks clear details, but still retains the general shape and layout. The costumes are compressed in a more uniform and less textured appearance.

All reconstructed images are noticeably less sharp compared to their originals. This is common in autoencoders, as they compress the input into a lower-dimensional latent space, losing some information in the process, which results in loss of fine details upon reconstruction. Despite the loss of detail, the general shapes and overall layouts are preserved, which indicates that the autoencoder is capable of capturing the broader structural aspects of the images, which are crucial for recognizing the objects.

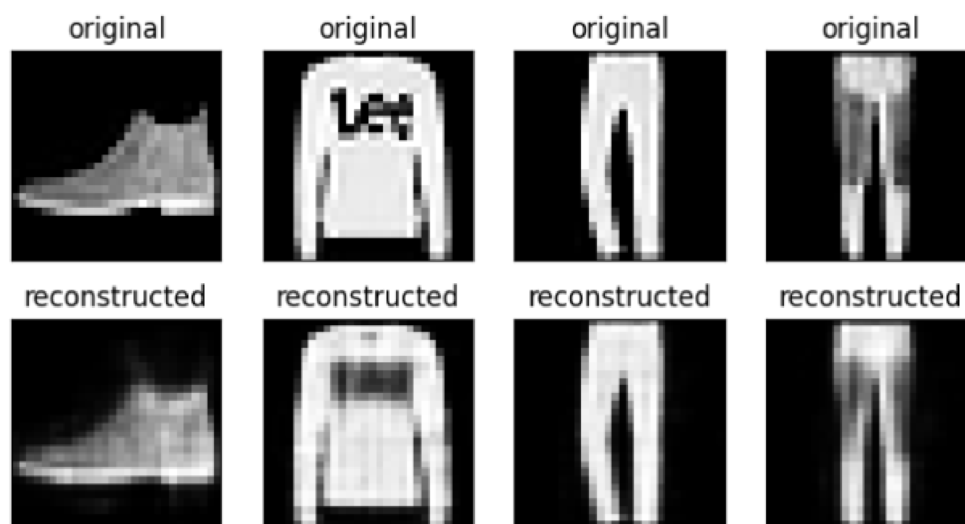


Figure 2.13: The original and reconstructed images from an autoencoder.

Overall, the autoencoder excels at compression and decompression tasks, learning functions directly from the input. This means that an autoencoder's performance and effectiveness are dependent on the training process. The model learns the most efficient way to represent that specific set of data in a compressed form, which is generally a lower-dimensional representation.

The networks above are all traditional artificial neural networks, they are well-designed for particular purposes with fixed input data types. However, you may find that they are difficult to deal with dynamic input/output data types and shifting goal functions. When the dimensions or types of the data change, these models must be rebuilt and trained again. In the following section, a new structure will be introduced to solve this limitation.

2.2.4 Transfer Learning and RayBNN

Humans thrive at transferring information between tasks. This implies that whenever we meet a new difficulty or task, we identify it and use what we've learned from past encounters. This makes our task easier and faster. For example, suppose a person knows how to ride a bicycle but is required to ride a motorcycle that he/she has never ridden before. In this situation, cycling skills will come in handy for activities such as balancing the bike, steering, and so on. This will make learning simpler than they would be for a total newbie. Such lessons are beneficial in real life since

they improve our skills and allow us to get more experience. Following a similar methodology, the term Transfer Learning [40] was introduced in the field of machine learning. This strategy entails leveraging information gained in previous activities to address a problem in a related target task. This is beneficial when the later task is comparable to the first task or the data is insufficient for the second task. Using the learned characteristics from the first task as a starting point allows the model to learn the second task more quickly and efficiently. Traditional neural networks often need extensive retraining when the goal function changes. For example, changing from classifying types of vehicles to classifying types of animals in images requires retraining in such a network. In contrast, transfer learning can quickly adapt the pre-trained model to new tasks by retraining only the final layers, which are typically responsible for decision-making based on the features extracted by earlier layers. Training a neural network from scratch on a completely new task every time is computationally expensive and time-consuming. Transfer learning reduces this burden by reusing learned features, thereby saving computational resources and reducing training times significantly. When goal functions shift, transfer learning allows the retention of useful features from the original model, ensuring that not all previous learning is discarded. It helps in preserving knowledge that the network has previously learned, which is a solution to catastrophic forgetting [20] in neural networks that refers to a neural network's tendency to completely forget previously learned information upon learning new information.

Base on the strengths of transfer learning, Ref. [42] presented a novel transfer learning model through 3-D emulation of real biological neural networks (BNN), which is called three-dimensional ray-traced biological neural network (RayBNN). Unlike the traditional ANNs, which are usually planar, RayBNN is built by uniformly distributing hidden neurons and glial cells within a 3-dimensional neural network sphere, guaranteeing that they do not overlap. As shown in Fig. 2.14, the model adopts glial cells (red spheres), neurons (green spheres), and the axons (green lines). The information learning is more efficient because the neurons in RayBNN are connected to each other randomly instead of in well-defined layer structure.

The physical architecture of a neural network is formed by assigning three-dimensional places to glial cells and neurons. The process starts with a random allocation of cell placements, which may cause overlaps or crossings between neurons and glial cells. Such intersecting cells are detected by the model and then eliminated to avoid structural inconsistencies. After the hidden neurons and glial cells have been positioned,

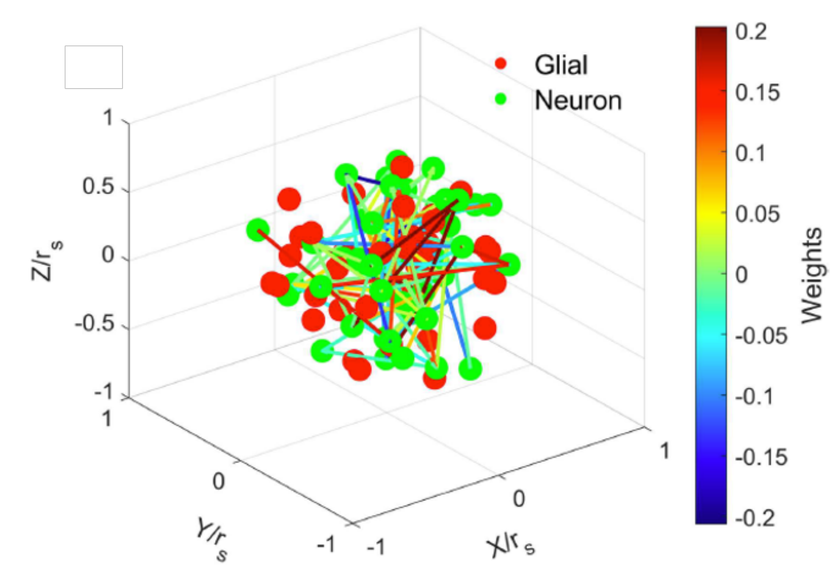


Figure 2.14: Stimulated 3d ray traced biological neural network (RayBNN) [42].

the positions of both the input and the output neurons can be established. In cases when the datasets include picture inputs, the input neurons can be assigned equally around the sphere's surface to maintain pixel distances. In contrast, the output neurons are fixed in the center of the sphere, replicating the arrangement of the human brain. The central position of output neurons allows the pooling and integration of information from hidden neurons since neural pathways converge at the sphere's center. Ray tracing is then used to generate new neural connections based on cell locations and radii, resulting in a realistic map of neuron-to-neuron communication paths. Every neural link in the network is represented as a sparse weighted adjacency matrix. This encoding captures the strength and structure of relationships in a space-efficient way.

Along with structural encoding, each neuron is assigned a Universal Activation Function (UAF), which allows the neuron's activation response to adjust over time [42]. This is especially beneficial during learning stages and information transfer. The network's operational stages consist of two passes: forward and reverse. The weighted adjacency matrix is used in the forward pass to determine the output of the neural network. In contrast, the backward pass computes the gradient of the weights, which are subsequently changed via gradient descent methods to improve the network's performance. In transfer learning, when the dataset changes, the network's architecture is constantly modified. This involves eliminating unneeded neural

connections and pruning excess neurons to keep the network streamlined and efficient for the new data context.

Fig. 2.16 is the algorithm of the RayBNN model. All output neurons are assigned to the center of the network, and the input neurons are assigned to the surface of the sphere. To maintain the order of the input data, input neurons are mapped onto the sphere's surface. Fig. 2.15(a) depicts how to organize the input neurons on the surface to maintain the original order of one-dimensional, two-dimensional, and three-dimensional data characteristics using a systematic mapping strategy. In addition, each neuron is assigned an equal solid angle on the sphere's surface, guaranteeing that all input neurons have unbiased communication with the hidden neurons underneath. When adapting the model to a new dataset with more dense input dimensions, then new neurons should be inserted between the existing ones, as shown by the red dots in Fig. 2.15(b), without having to move the original neurons. Alternatively, when more data features need to be added to the existing ones, as shown in Fig. 2.15(c), the old neurons can be shifted to the sphere's northern hemisphere, making room for new neurons in the southern hemisphere while maintaining the original order of the data features [42].

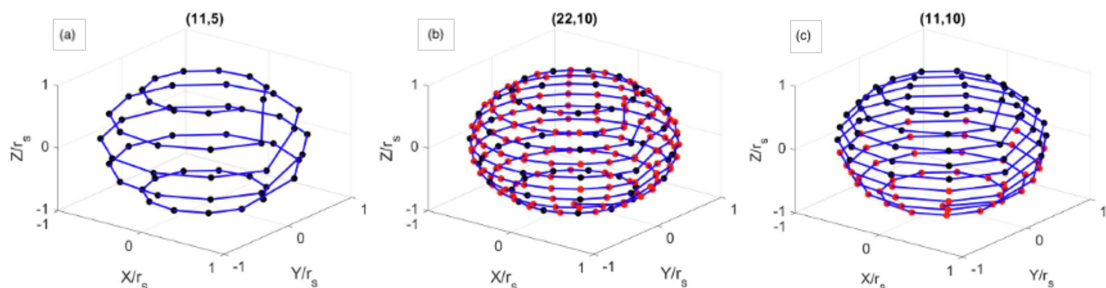


Figure 2.15: *(a) Location of input neurons at the surface of the network sphere. (c) Transfer of input neurons to a new network sphere where the dimension of the data is densified. (b) If the new dataset concatenates the old dataset, then the old neurons migrate to the north while new neurons are created in the south of the new network sphere. [42].*

The dimensions of the datasets may vary when transferring knowledge between datasets. The quantity of the input and output neurons are changed according to the variation of datasets. Meanwhile, intersecting cells are all deleted during the assignment when the cells are overlapped. Ray tracing algorithms are then used for generating the connections among neurons using the positions and radii of the

cells. RayBNN uses ray tracing methods to create unidirectional connections between neurons. The most interesting part of this method is that authors use 3 different kinds of raytracing (RT) algorithms to connect neurons: in RT algorithm 1 (RT-1), each neuron generates K rays with random angles and limitless lengths; to improve efficiency, they then developed RT algorithm 2 (RT-2) to directly link rays so that it connects each neuron in the neural network by a finite-length ray; and last but not least, RT algorithm 3 (RT-3) was proposed, which builds on the prior method and assumes that long-distance connections may be disregarded, and rays with little distance are directly linked. These connections occur only when the neurons are in direct line of sight of each other. Glial cells play an important function in preventing connections between neurons that are too far apart. This method helps to reduce overfitting in the learning model.

After that, the neural connections will be mapped into the weighted adjacency matrix. This storing approach provides RayBNN the ability to transform into other topologies without changing the size of the matrix. Moreover, a Universal Activation Function (UAF) was applied across all neurons. UAF is designed to evolve during information transmission, with increasing the plasticity of neurons' activation responses [42]. After the matrix is configured completely, the forward pass algorithms are used to update the input and the output state vectors from precious state vectors. A backpropagation algorithm is used to calculate the gradients for optimizing the parameters of RayBNN. Finally, useless neural connections and neurons are removed by the algorithm without influencing the performance.

By using RayBNN's adaptability, we can alter and move the network to meet any given architectural framework to overcome the inconvenience of traditional neural networks. For example, training large neural networks is usually a time-consuming operation. To overcome this problem, users can start with training a small-scale neural network that can learn quickly, and then transfer this obtained information to a much bigger network. This method effectively shortens the overall training duration. As the network grows throughout the transfer, more neurons are added to the three-dimensional network sphere. Similarly, we can also use ray tracing to create new connections while retaining old ones. If required, the size of the sphere is changed to provide a consistent rate of neuron non-overlap. When presented with a fresh dataset that requires a decrease in neurons and connections, RayBNN may selectively delete connections, prioritizing the elimination of those with the smallest absolute weights since they contribute the least. To improve the efficiency and accuracy of

Algorithm 1: Overview Algorithm for RayBNN

```
function RayBNN(Model, λ, Dataset):
    //Initialization
    Data ← Dataset(λ);
    Model(λ).initNetworkSphere(Data);
    Model(λ).raytraceConnections();
    Model(λ).createWAdj();
    while true do
        //Training the Model
        Loss ← ∞;
        while !isPlateau(Loss) do
            Model(λ).forwardPass(Data);
            Model(λ).backwardPass(Data);
            Loss ← Model(λ).crossValidation(Data);
        end while
        //Transfer Learning
        Model(λ+1) ← Model(λ);
        λ ← λ + 1;
        Data ← Dataset(λ);
        Model(λ).delConnections();
        Model(λ).delUnusedNeurons();
        Model(λ).addNeurons();
        Model(λ).raytraceConnections();
    end while
```

Figure 2.16: Overview Algorithm for RayBNN [42].

the networks, certain neural connections are reallocated to other neurons during the pruning process. Neurons that become redundant are then eliminated, resulting in a more efficient network with better performance and accuracy.

2.3 Overfitting and Underfitting

When training models, the situation that high accuracy on the training set but significantly lower accuracy on the validation set is called “overfitting”. It occurs when the model structure is too complex or the training data is too small for models to learn from, the models can memorize the training data instead of generalizing it. On the contrary, poor accuracy on both training and validation sets is called “underfitting”. It occurs when the model structure is too simple or the training time is not enough, the models may not be able to learn from the data comprehensively.

To overcome overfitting, one may reduce the depth of the neural networks or add some noise to the data. By introducing noise, the model is forced to learn more generalizable features across different samples. It makes the model not only perform well on slightly different data samples but also ensure that the small variations in input data do not lead to significant changes in output. For underfitting, increasing the model complexity with additional layers, or increasing training duration to give the model more time to learn the patterns in the data can help.

2.4 Evaluation metrics

For this project, we used accuracy, precision, recall, and F1-score to measure the performance of our models, which are shown in the following content:

1. **Accuracy**: It measures the overall correctness of the model and is calculated as the ratio of correctly predicted instances (both positive and negative) to the total instances.
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$
2. **Precision**: It measures the proportion of correctly predicted positive instances among all instances predicted as positive.
$$\text{Precision} = \frac{TP}{TP + FP}$$
3. **Recall**: It measures the proportion of correctly predicted positive instances among all actual positive instances.
$$\text{Recall} = \frac{TP}{TP + FN}$$
4. **F1-Score**: It is the harmonic mean of precision and recall, providing a balance between the two metrics.
$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Within formulas above:

(a) **True Positives (TP)**: The number of positive instances correctly predicted as positive.

(b) **True Negatives (TN)**: The number of negative instances correctly predicted as negative.

(c) **False Positives (FP)**: The number of negative instances incorrectly predicted as positive.

(d) **False Negatives (FN)**: The number of positive instances incorrectly predicted as negative.

For balanced datasets, accuracy, precision, recall, and F1 score that mentioned above can evaluate the models and help us to understand the performance of a model comprehensively from different aspects. For those imbalanced dataset, we added Cohen's Kappa Coefficient [4] for measuring the performance of our model. Cohen's kappa coefficient (κ) is a statistic used to measure inter-rater reliability for qualitative (categorical) items. It is also applied to evaluate classification models in machine learning, where it can compare the agreement between the model's predictions and the true labels of the data.

The expression for Cohen's kappa is:

$$\kappa = \frac{p_o - p_e}{1 - p_e}$$

where:

- p_o is the relative observed agreement among raters. In model evaluation, it represents the agreement between the model's predictions and the true labels.

- p_e is the hypothetical probability of chance agreement, calculated based on the marginal probabilities of each rater (or class in machine learning) assigning labels randomly.

The kappa value (κ) ranges from -1 to 1. A value of 1 indicates perfect agreement, 0 indicates the amount of agreement that can be expected by chance, and negative values indicate less-than-chance agreement.

Due to its characteristics and formulas above, when the data features a single class, Cohen's Kappa value defaults to zero. For example, even though the algorithm identifies all the data correctly, which means a subject is always in a single state,

the true labels are 100 percent in this state, where p_o is 1 (or 100 percent). The algorithm also classifies every instance as this state, which means there is only one class, the probability that a random classifier would predict (p_e) is also 1 (or 100 percent), because there is no other choice. Thus, we have $p_o = 1$ and $p_e = 1$, and $\kappa = 0$.

The result is undefined, but the probability of chance agreement in statistics is 1, so the Kappa value is set to be 0. This can be explained by the Kappa value measures the agreement that is above and beyond what is expected by chance, therefore, no room is available for improvement due to the lack of variability in the dataset. The Kappa statistic presents as low when all of the instances are in one category, which is known as the Kappa paradox. This paradox will not occur if we have several classifications.

Chapter 3

Datasets and Methodology

The first part of this chapter introduces datasets that are used in this project. The methodologies of CNN to RayBNN, the hardware and the framework that is used to train these models are described at the second half of this chapter.

3.1 Datasets

3.1.1 MNIST dataset

The MNIST (Modified National Institute of Standards and Technology) dataset [5] is a large database that contains a 70,000 of handwriting digits. The MNIST dataset was originated from the NIST (National Institute of Standards and Technology)'s [10] original sets. It provides a more balanced and acceptable dataset for machine learning studies in contrast to the initial NIST training set, which was drawn from American Census Bureau employees, and the test set was collected from the handwriting of American high school students and did not provide an appropriate combination and a matched requirement for intensive machine learning investigation. In addition to re-mixing the NIST database, the switching from the NIST to the MNIST database required standardization, or down-sampling of the original black and white pictures. During this procedure, the pictures were anti-aliased, which introduced varied shades of gray. The incorporation of this preprocessing procedure not only improves data homogeneity but also simulates the intricacies seen in authentic handwritten characters.

Each image in MNIST dataset is a 28×28 pixel grayscale image. In the original binary format of the MNIST dataset, each pixel is represented as a single byte (0

to 255) indicating the grayscale intensity. Besides, it is labeled with the digit it represents. Fig. 3.1 displays examples of the images.

Figure 3.1: Image examples in MNIST dataset [8].

Machine learning models, particularly those involved in image recognition tasks, are often evaluated on their performance with the MNIST dataset, such as CNN models. The simple and clean format of the dataset allows for straightforward comparisons between different algorithms. MNIST is widely used in educational settings to help beginners understand the basics of machine learning techniques.

3.1.2 MWT dataset

Maintenance of Wakefulness Test(MWT) dataset is created in the process of research on excessive daytime sleepiness (EDS) [6], which is a significant issue characterized by persistent drowsiness and a lack of energy throughout the day, even after getting enough sleep at night. People with EDS struggle with uncontrollable urges to nap, impacting their ability to function normally during waking hours. EDS is a condition that often occurs among patients and those people whose sleep decreases in a long-term [11], which is defined as the inability to remain awake and alert during the main waking hours of the day and to sleep unintentionally or at inappropriate times almost daily for at least 3 months. To learn more about the EDS condition statistically,

objective evaluation methods of sleep and wakefulness occur to help. Maintenance of Wakefulness Test (MWT) is generated as one of the assessment tools to evaluate the performance of sleep and wakefulness [27, 26]. It is the most commonly used test to assess the ability to sustain alertness in the context of increasing sleep pressure.

The MWT is typically administered after the Multiple Sleep Latency Test (MSLT), which measures how quickly a person falls asleep after periods of wakefulness. The data in MWT dataset is collected from individuals with EDS. It contains the electrical brain activity called electroencephalogram (EEG), eye movements called electroocoulogram (EOG), and muscle activity called electromyography (EMG) during sleep. The main measure of the MWT is the mean sleep latency time, which is generally considered abnormal if it is smaller than 8 min. Remaining awake for at least 40 min in all four tests is objective evidence that the subject can remain awake [9].

In MWT, the Microsleep episode (MSE) state is regarded as an objective indicator of EDS [15] condition, which is a brief sleep episode that is shorter than 15 seconds [37], recorded based on occipital EEG derivations (O1-M2, O2-M1), EOG, and videography. Aside from wakefulness and MSE states, 2 kinds of EEG patterns exist, which are defined uneasily, and are classified as microsleep episode candidates (MSEc) that are defined as not meeting all MSE requirements, and episodes of drowsiness(ED) that are defined as more unclear wake or MSE or MSEc states, for example, eyes closes more than 80 percent [15]. Therefore, the 4 stages of MWT are Wake, MSEs, MSEc, and ED.

In MWT dataset, by capturing simultaneously standard EEG, EOG, submental EMG, EKG (with electrodes on the right subclavicle and left thorax at heart height), respiratory flow, and facial videography with audio, the recording process was comprehensive. Placement of EEG electrodes conformed to the established 10–20 system [17] as shown in Fig. 3.2, targeting locations O1-M2, O2-M1, C3-M2, C4-M1, CZ-M1, F3-M2, and F4-M1, all referenced to opposite mastoid sites, ensuring a thorough neurophysiological mapping, so that we can get consistent and replicable EEG recordings data. This system can be used in various research, such as EEG exams, sleep studies, and clinical studies.

The RemLogic devices were utilized for data acquisition for this dataset, with a high-frequency collection and archiving rate set at 200 hertz. This data was subjected to a trio of hardware filters—specifically, a high-pass filter at 0.3 hertz, a low-pass filter capping at 70 hertz, and a 50-hertz notch filter to mitigate interference. Post-collection, the data was formatted into the European Data Format (EDF)

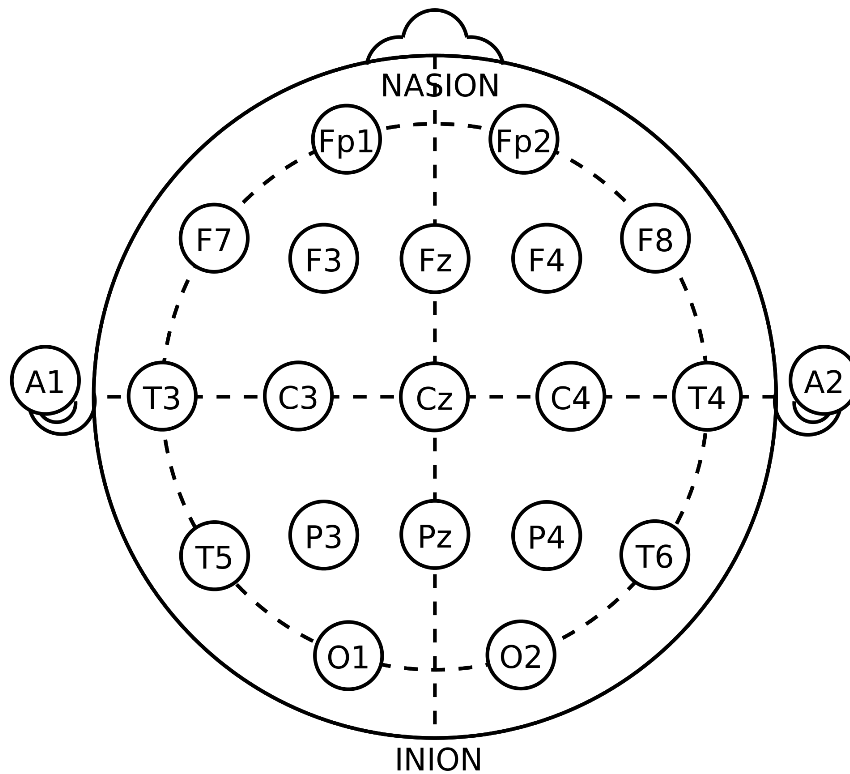


Figure 3.2: Electrode locations of International 10-20 system for EEG (electroencephalography) recording [25].

to streamline subsequent analysis stages.

As shown in Table 3.1, 76 patients' EDS data was recorded by taking MWT from Jelena Skorucak's team [37]. 53 of the patients' data was stored to be the training dataset, 12 patients' data was stored to be the validation dataset, and the rest 11 patients' data was saved as the testing dataset. Detailed information about patients' numbers, the proportion of males and females, mean ages with standard deviation, and the proportion of time that each state (wake, MSEs, MSEc, and ED) spends during the experiment are denoted in this table. It is worth noticing that the percentage of different data types in the dataset varies significantly. The proportion of wake condition is the highest among all data while MSEc condition has the lowest percentage.

The MWT requires accurate identification of sleep stages from EEG, EOG, and EMG recordings, which are also called polysomnography recordings. As a result, human experts' scoring is introduced [15]. Well-trained sleep technologists or sleep physicians are skilled in interpreting these recordings. This interpretation is criti-

	Training	**Validation**	**Testing**
Number	53	12	11
Male/Female	35/18	6/6	9/2
Age (mean $\pm$ SD years)	45.99 $\pm$ 18.17	44.64 $\pm$ 20.56	44.92 $\pm$ 14.48
Fraction of time in Wake	0.89	0.85	0.91
MSEs	0.08	0.09	0.05
MSEc	0.01	0.01	0.01
ED	0.02	0.05	0.03

Table 3.1: Dataset Distribution for training, validation, and testing.

cal to determine the exact moment sleep begins, differentiating between wakefulness, light sleep, and deeper sleep stages. While some aspects of polysomnography can be analyzed automatically by software, human oversight is essential due to the subjectivity involved in interpreting certain signals. For example, the transition between wakefulness and sleep can be subtle, with small changes in EEG patterns that might be missed or misclassified by automated systems. Sleep onset can sometimes involve ambiguous signals, which can resemble wakefulness in a sleepy individual. Experts can discern these patterns better than current automated systems. In cases where the patient's sleep architecture is unusual or disrupted by medical conditions, human experts can provide insights that are not easily derived from automated processes. They can interpret complex scenarios where multiple sleep-related factors and disorders intersect.

The scoring of the MWT dataset was conducted by an experienced scorer [15] and in around 2/3 of the trials, the final scoring was verified by other experienced scorers and differences were resolved by discussions. Wakefulness, MSEs (visible in both channels), microsleep episode candidates (MSEc), or episodes of drowsiness (ED) were scored as defined in Bern continuous and high-resolution wake-sleep (BERN) scoring criteria [15], which were developed to provide a standardized approach to determining sleep onset during the MWT. Under the BERN criteria, sleep onset is defined not just as the first epoch (30-second period) of any sleep stage, but specifically requires the presence of 15 consecutive seconds of cumulative sleep within a 30-second epoch. This criterion is intended to ensure that brief and potentially ambiguous instances of sleep or drowsiness are not overinterpreted as definitive sleep. This scoring method is used to increase the specificity of the maintenance of the wakefulness test, reducing the likelihood of false positives in determining sleep onset. It is particularly useful in clinical settings where precise assessment of wakefulness is necessary for evaluating

the safety of individuals in jobs requiring high alertness or for assessing the efficacy of treatment in sleep disorders.

3.2 Methodology

3.2.1 CNN+ANN approach with RayBNN

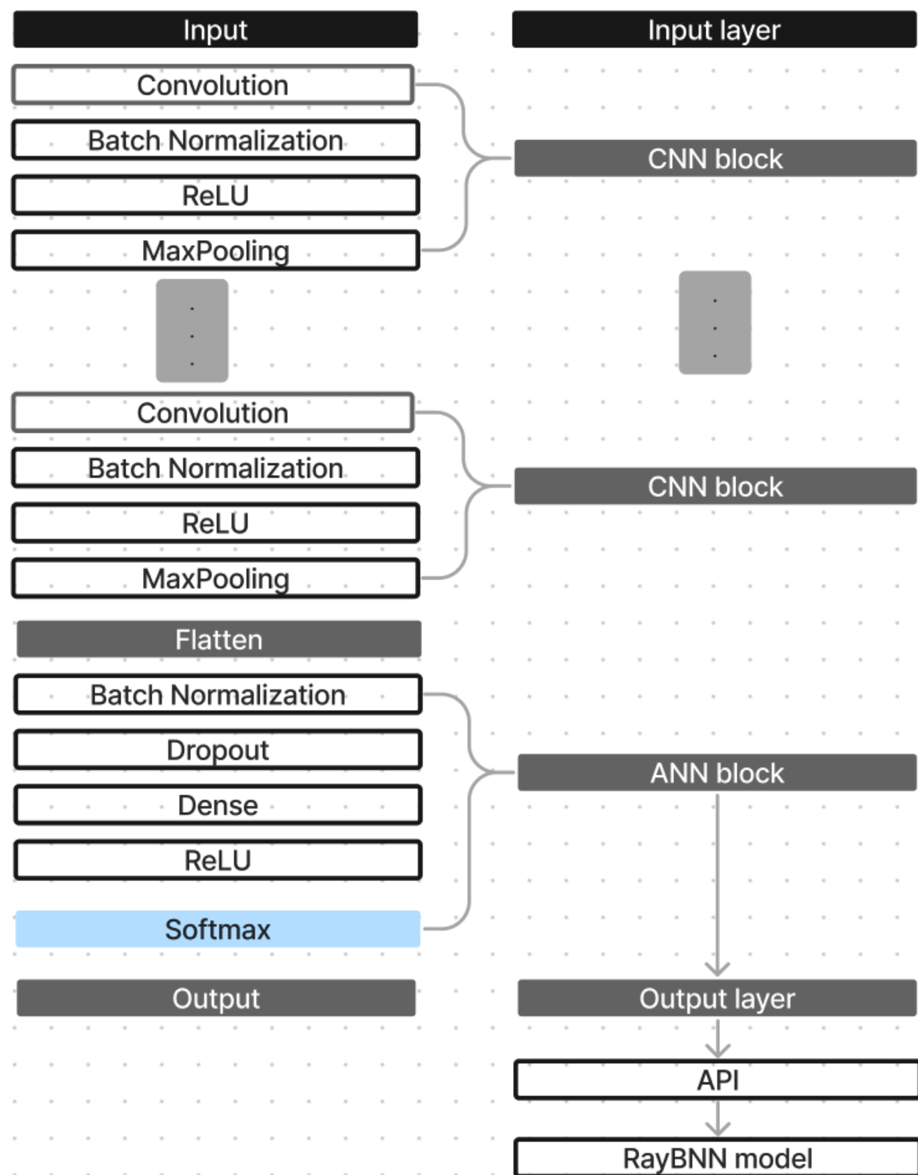


Figure 3.3: CNNs+ANN, and applied to RayBNN.

Fig. 3.3 demonstrates the hybrid neural network architecture that integrates CNNs with an ANN and applies it to RayBNN. This structure is designed to leverage the strengths of each type of network for data processing tasks.

The model begins with CNN blocks, which are designed for extracting features. The CNN blocks contain several layers including convolution layers that apply filters to the input data to extract features. Batch normalization layers normalize the outputs of convolutional layers to improve training stability and speed. The ReLU activation function introduces non-linearity, enabling the model to learn more complex patterns. After that, max-pooling layers are applied to reduce the dimensionality of the data, which helps to decrease computational costs and prevent overfitting. The output from the max-pooling layers is flattened into a one-dimensional array to be prepared for processing in the dense layers of the ANN.

Following the CNN blocks, the structure incorporates ANN layers that further process the features extracted by the CNN. The ANNs include dense layers, which are fully connected and play a key role in classifying the data based on the learned features. Dropout layers are set before the dense layers to mitigate overfitting by randomly dropping units during training. Additional batch normalization and ReLU layers are applied to enhance the learning process and ensure effective forward propagation of data.

The final layer in the ANN sequence is a softmax layer, which is used for multi-class classification, outputting probabilities that sum to one, making it easy to determine the class by selecting the highest probability. We stored the label and feature results of the max-pooling layer. This extracted and processed information is then passed via an API, regarded as the input data of the RayBNN model. The API serves as a crucial link, facilitating the flow of processed data between the CNN structures and the RayBNN. The RayBNN model utilizes these inputs to perform more complex data interpretations or decision-making tasks. This part of the model is used to provide flexibility and adaptability in handling diverse and dynamic datasets.

3.2.2 CNN+Autoencoder approach with RayBNN

Fig. 3.4 depicted a structure that integrates a CNN with an autoencoder model and then applies it to the RayBNN. The encoder consists of several CNN blocks. Max-pooling layer is not included in the final CNN block to capture more detailed representations for better reconstruction quality. The decoder architecture has several trans-

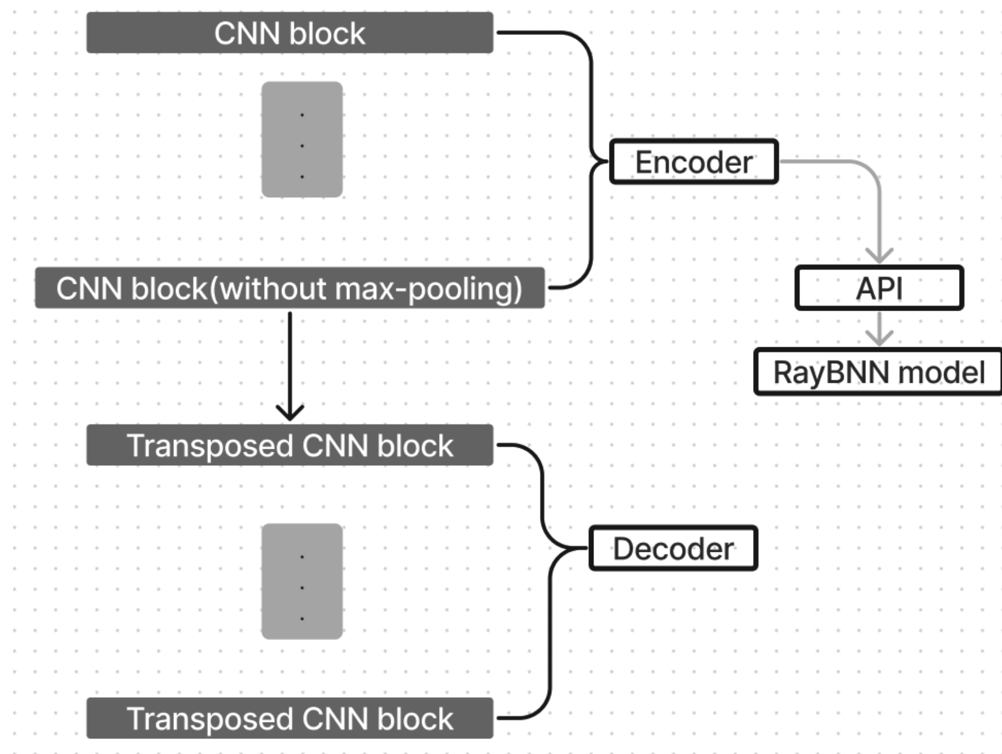


Figure 3.4: CNN Autoencoder+RayBNN model structure.

posed CNN blocks. Each CNN block contains a transposed convolutional layer with a ReLU. The transposed convolution layer, also known as the deconvolution layer, aims to reconstruct the compressed representation from the encoder. For example, Fig. 3.5 illustrates how a transposed convolution layer works. The left sub-image presents the input feature, and the transposed convolution layer contains a kernel size of 3×3, and a stride size of 2. The input dimension is 2×2 in height and width. For each element in the input feature, the filter spread over the output feature with a stride of 2, and added to the corresponding positions in the output. The input feature is expanded according to the stride, resulting in gaps that will be filled by the filter. The output, as shown in the right sub-image, is the regions where the filter overlaps are summed. This operation increases the spatial dimensions of the output.

The transposed CNN block contains a transposed convolutional layer with ReLU. This part of the network works to reconstruct the input data from its encoded and compressed representation. It performs upsampling with learnable parameters, allowing the network to learn how to upscale the features, which is crucial for effectively rebuilding the input data with high quality.

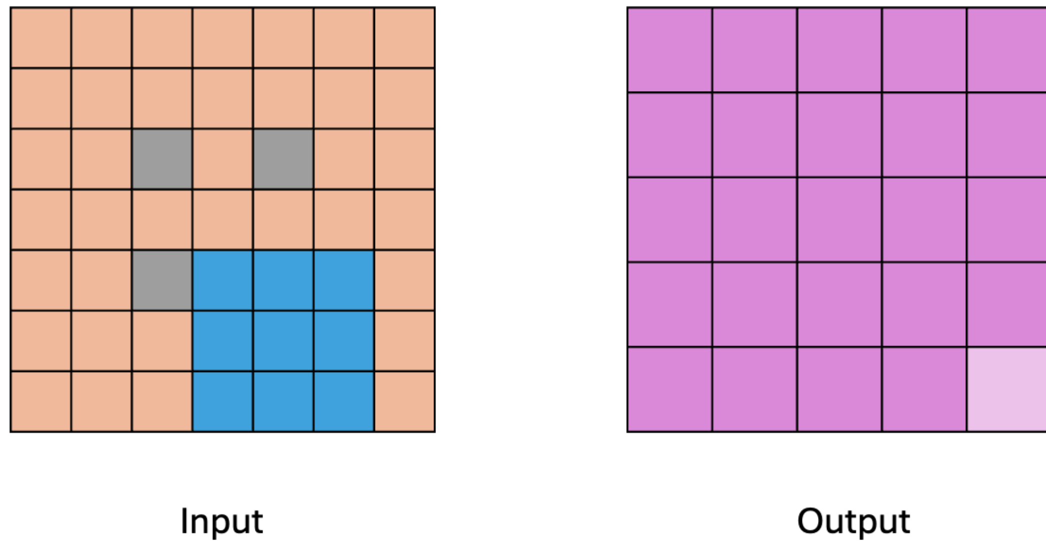


Figure 3.5: Transposed convolution working process.

After training, the features of the whole dataset will be extracted through the encoder, which is further fed to RayBNN API for processing.

3.3 Preparation for RayBNN API

After the data features were extracted, they were input into the RayBNN. The RayBNN model is implemented with Rust, and an application programming interface (API) was developed in Python.

3.4 Hardware

Due to the need for complex computation, large dataset requirement, and speed, a high-performance computing (HPC) cluster Narval on the server of Digital Research Alliance of Canada was used for the work. The Narval cluster provides researchers with access to advanced computing infrastructure, including powerful CPUs, GPUs, and high-speed storage systems (as shown in Table 3.2), to facilitate complex computations and data-intensive research projects. Narval cluster provides distributed deep learning frameworks such as Pytorch. Its GPU resources are instrumental in accelerating the process of neural network models, reducing computational time significantly.

nodes	cores	memory	CPU
1145	64	249G	2 x AMD Rome 7532 @ 2.40 GHz 256M cache L3
33	64	2009G	2 x AMD Rome 7532 @ 2.40 GHz 256M cache L3
3	64	4000G	2 x AMD Rome 7502 @ 2.50 GHz 128M cache L3
159	48	498G	2 x AMD Milan 7413 @ 2.65 GHz 128M cache L3

Table 3.2: Cluster Specifications.

The hardware and software requirements for RayBNN are as follows.

1. System requirements
   (a) RTX 3090 or more powerful with at least 24GB VRAM
   (b) 32GB RAM
   (c) 30 GB of disk space
   (d) Docker(https://www.docker.com/)
   (e) Rust(https://www.rust-lang.org/)
   (f) Arrayfire (https://github.com/arrayfire/arrayfire)
   (g) Arrayfire Rust (https://github.com/arrayfire/arrayfire-rust)
   (h) Pytorch (https://pytorch.org/)
   (i) Pytorch geometric (https://github.com/pyg-team/pytorch-geometric)
   (j) Matlab

3.5 Framework

This project is implemented in Python 3.11 using Compute Unified Device Architecture (CUDA), cuDNN and scipy-stack. CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general-purpose processing – an approach termed GPGPU (General-Purpose computing on Graphics Processing Units). cuDNN is a GPU-accelerated library for deep neural networks, which serves as a higher-level API to CUDA and is specifically designed to provide highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. It is part of the NVIDIA Deep Learning SDK. The SciPy stack refers to a collection of open-source

software for mathematics, science, and engineering, predominantly Python-based. It typically includes core packages like NumPy, SciPy library, Matplotlib, IPython, Sympy, and Pandas. These libraries are designed to work well together and are often installed collectively for scientific computing tasks. We used open-sourced software libraries like TensorFlow and Keras for experimentation with models. Keras can be run on top of TensorFlow, allowing a more user-friendly interface to complex operations.

Chapter 4

MNIST dataset

This chapter characterizes the model with the MNIST dataset. It first introduces dataset arrangements for training and testing, followed by the results and discussions.

In this project, we adopted a 10-fold test scheme, in which the MNIST dataset was randomized and split into 10 sections. During each fold, 8 sections for training and 2 sections for testing.

In this dataset, we first adopted the CNN+ANN model proposed in Ch. 3. In this model, we adopted two CNN blocks. Both blocks have the kernel size of 3×3 with a stride of 1. The max-pooling layer in each block reduces the spatial dimensions of the feature by half by using a 2×2 pooling size. It is proved that stacking two 3×3 convolutional layers can achieve a better performance in receptive field than a 5×5 convolutional layer, which was prominently highlighted in the Visual Geometry Group(VGG) network's design [36]. Because in the same number of filters in both configurations, using a single 5×5 convolutional layer contains 5×5(equals to 25) weights per filter, however, using 2 3×3 convolutional layers contain 3×3×2(equals to 18) weights per filter. Thus, using 2 3×3 convolutional layers makes the network more efficient in parameters. Meanwhile, stacking 2 smaller convolutional layers can introduce more nonlinearity into the network by placing a ReLU activation function between the convolutional layers. This allows the network to obtain more layers that contribute to a deeper architecture, enabling the network to learn more complex features at each layer. In this model, we used 32 filters in the first CNN block and 64 in the second block to capture necessary features.

After CNN, the data features are flattened and fed to a 1024-neuron flattened layer and enter ANN with 512 hidden neurons. The output layer consists 10 neurons, corresponding to the number of output classes in MNIST. With the approach of

CNN+RayBNN, we extracted and stored the features from the last CNN block, and then applied them to RayBNN, which is introduced in chapter 3, thereby will not be introduced in details here.

In a second test, we implemented the CNN with an autoencoder approach as described in the previous chapter. In this model, the auto-encoder contains 3 CNN blocks. The 1st CNN block performs a 2D convolution with 16 filters, each having a kernel size of 3×3 and a stride of 2. Similar to the 1st block, the 2nd CNN block performs with 32 filters using the same kernel and stride settings as the 1st layer. The last CNN block, which has Max-pooling layer removed, has 64 filters with a larger kernel size of 7×7 to preserve the spatial dimensions while allowing the final convolutional layer to perform better feature extraction without further downsampling, keeping the quality of the reconstruction in the decoder. After the model is trained, we extracted features from the last CNN block, and applied them to the RayBNN model.

The decoder primarily uses transposed convolutional blocks, which operate in a way that essentially upsamples the feature, gradually increasing their spatial resolution. The transposed convolutional layer in the block is accompanied by ReLU activations and also have same kernel sizes and strides as the numbers in encoder correspondingly. In the last transposed CNN block, it decompressed to one grayscale output channel of the MNIST images to match the format of the original input images.

4.1 Results

4.1.1 Feature representations of the models

CNN+ANN approach

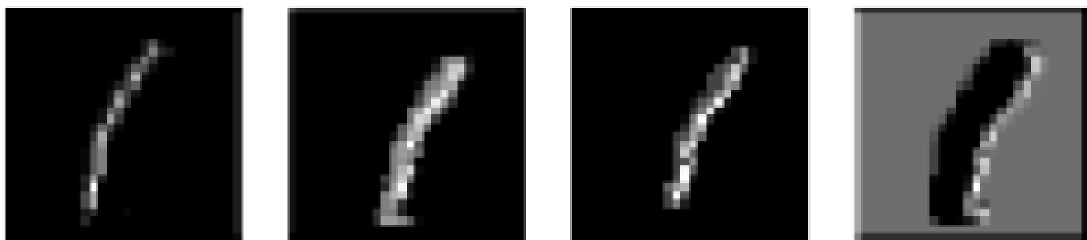


Figure 4.1: Feature representation in the first convolutional layer.

Fig. 4.1 demonstrates the feature representation of the digit "1" processed by the first convolutional layer. These representations are relatively detailed, highlighting various specific features such as edges and curves of the digit "1". Each image represents a different filter's response to the same input image. The variety in the representations suggests that different filters are tuned to detect different features, some focusing on the main vertical stroke of the digit "1", while others might pick up details or variations in stroke intensity and alignment.

Figure 4.2: Feature representation in the first max-pooling layer.

Fig. 4.2 presents the feature output of digit "1" after the first max-pooling layer. The sub-image on the left contains a more compact figure of the digit, and the sub-image on the right has a small feature at the bottom left of the figure. The spatial resolution of each feature is reduced, resulting in more abstract representations of the detected features. Max-pooling enhances the dominant features by reducing noise and small variations in the feature maps, focusing on the maximum value. This reduction sharpens the focus on essential elements like the vertical core of the "1", smoothing out less relevant details and making the network less sensitive to small positional changes and variations.

Fig. 4.3 is the feature representation after the second convolutional layer. With a further set of convolution operations, these features begin to combine and further abstract the basic features identified by the first layer into more complex patterns. The filters in this layer can capture higher-level features, which represents combinations of edges and curves. The representations are less interpretable than those from the

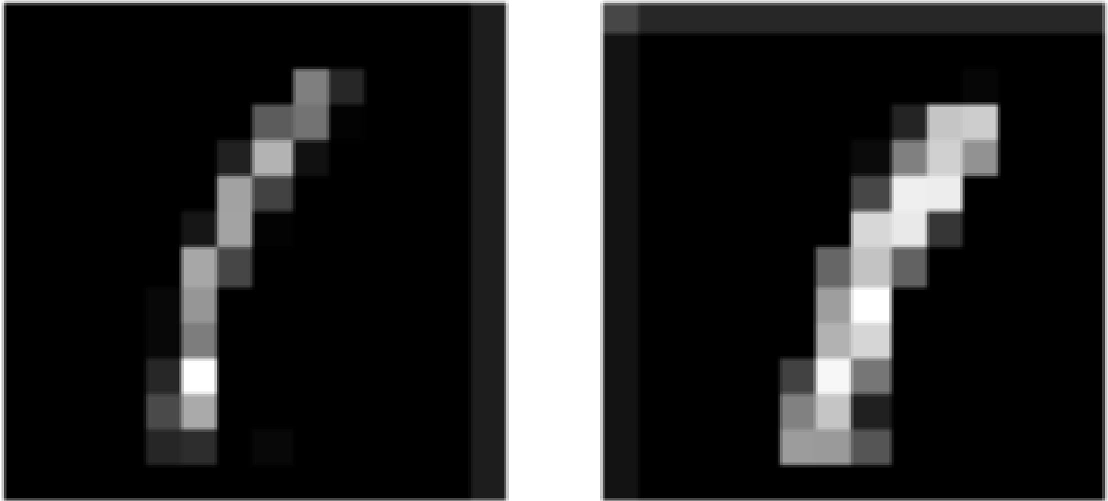


Figure 4.3: Feature representation in the second convolutional layer.

first convolution layer, indicating an increased level of abstraction. They are crucial for understanding higher-level structures within the digit, potentially preparing for classification.

Figure 4.4: Feature representation in the second max-pooling layer.

Like the first max-pooling layer, the second max-pooling layer further reduces the spatial resolution and abstracts the feature representation, as shown in Fig. 4.4. It emphasizes the most significant aspects of the input, even more, reducing the size and focusing on critical regions of the feature maps. At this stage, the features are

highly abstracted and geared towards helping the network make decisions based on the overall structure and presence of key features rather than specific details.

Above images present the feature representations in convolution and max-pooling from shallow to deep layers. This transformation from detailed to abstract representations allows the network to effectively classify inputs by focusing on the most informative features, reducing the influence of noise and irrelevant details.

CNN+Autoencoder approach

Figure 4.5: Reconstructed output of CNN autoencoder model.

Fig. 4.5 displayed several original input (the digits on the first line) and reconstructed output (the digits on the second line) that were processed by the CNN+Autoencoder model. The output digits are less clear than the input ones, losing some details, such as, the tail at the bottom of digit "7" in the third sub-image of the input is lost when observed in the output. The reconstructed images exhibit a little blurring. This is because the autoencoder can not perfectly decode all the nuances of the data from its compressed representation. However, the overall layout of the digits is all kept in the output using convolutional layers inside the encoder/decoder, which indicates that the convolutional layers present their capability of extracting features in input data.

4.1.2 Comparisons and analysis

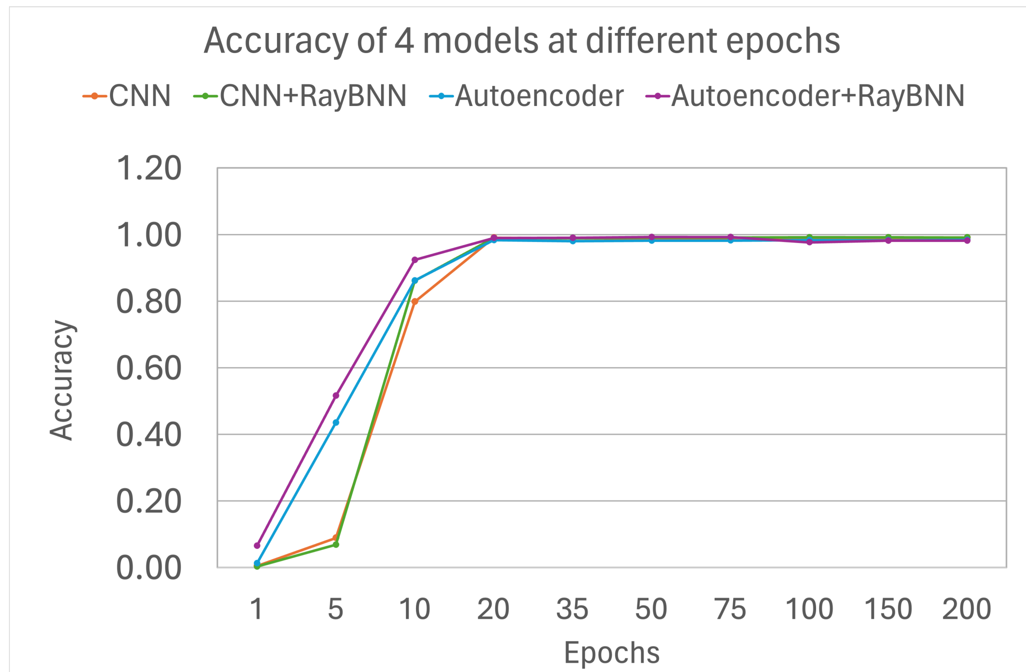


Figure 4.6: Accuracy of four models at different epochs.

In Fig. 4.6, the CNN model starts at an accuracy of 0.0500 at epoch 1 and experiences a steep increase to approximately 0.7984 by epoch 10, stabilizing and slightly improving to 0.9910 by epoch 200. This indicates a rapid learning rate initially, which plateaus of the model approaches its performance limit. The CNN+RayBNN model shows a similar trend but starts at a slightly lower accuracy of 0.0310 at epoch 1 and ends with a marginally lower accuracy of 0.9915 at epoch 200. The addition of RayBNN appears to slightly delay the initial learning speed but achieves a comparable performance by epoch 200. The Autoencoder starts at 0.0013 accuracy at epoch 1 and shows a significant jump to 0.8620 by epoch 10, achieving a peak accuracy of 0.9840 by epoch 200, which is slightly lower than the CNN models, indicating that while powerful, the Autoencoder alone may not capture all the predictive patterns. The Autoencoder+RayBNN begins with a higher initial accuracy of 0.0652 at epoch 1, soaring to 0.9240 by epoch 10 and peaking at 0.9819 by epoch 200. The combination of Autoencoder and RayBNN showcases a synergy that allows for a higher starting accuracy and rapid early improvement.

We can observe distinct learning curves and performance stabilization points across the different metrics. The CNN model consistently shows significant improve-

ment up to epoch 50, where its metrics begin to plateau, indicating that the model is approaching its learning capacity. It achieves its best overall performance by epoch 100, which is where the model's accuracy, precision, recall, and F1-score reach their highest values simultaneously. This performance stability makes the CNN model a dependable choice for classification tasks. Beyond this point at epoch 100, the model does not exhibit substantial gains, suggesting a thorough model optimization has been achieved by this epoch.

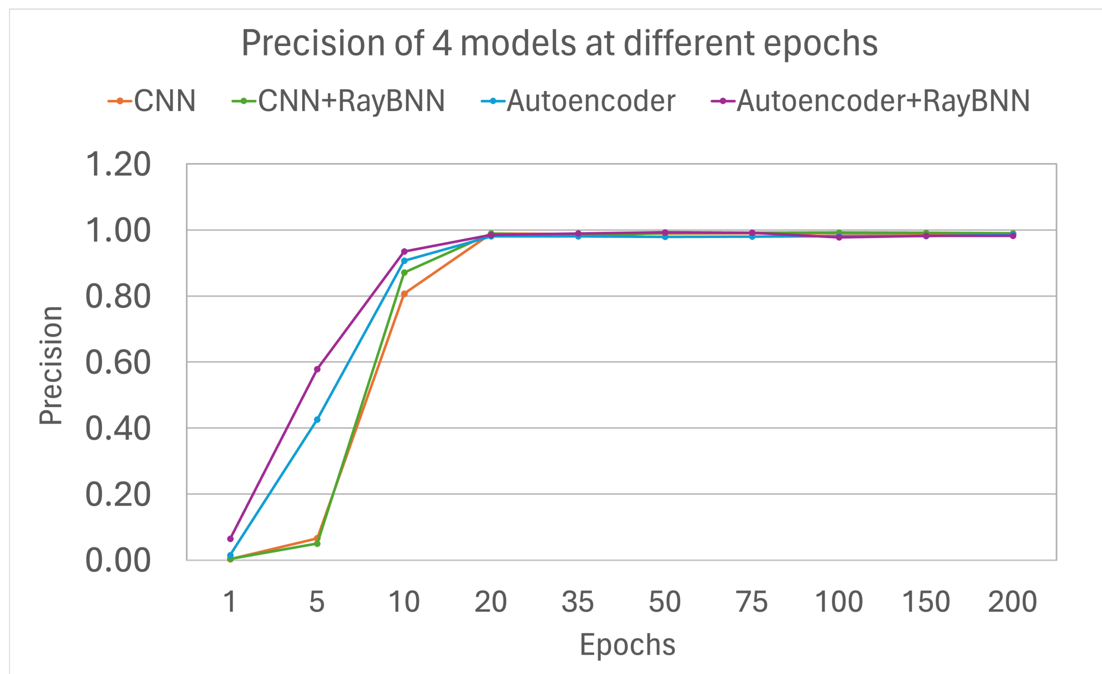


Figure 4.7: Precision of four models at different epochs.

From Fig. 4.7, the CNN model's precision begins at 0.0310 and reaches 0.8068 by epoch 10, slightly lower than its accuracy, indicating some false positives are present but improves and closely matches its accuracy by epoch 200 with 0.9887. Precision for CNN+RayBNN starts at 0.0039 and increases to 0.8712 by epoch 10, showing the highest precision amongst the models at this point, finishing at 0.9900 by epoch 200, which implies this model is particularly good at minimizing false positives when making predictions. The Autoencoder's precision starts higher at 0.0151, rises to 0.9070 by epoch 10, and ends with the highest precision of 0.9866 by epoch 200, suggesting a refined ability to make accurate positive predictions. Autoencoder+RayBNN begins with a precision of 0.0646, escalating to 0.9350 by epoch 10, and slightly decreases to 0.9825 by epoch 200. This drop could indicate overfitting or a limitation

in recognizing positive cases over time.

As shown from Fig. 4.6 to Fig. 4.7, the CNN+RayBNN converges faster than the CNN. At around epoch 35, this model swiftly ascends to its peak performance, particularly noticeable by epoch 100, where all its metrics are at their best performance. The rapid attainment of peak performance indicates that the synergy between CNN and RayBNN enhances the model's learning efficiency, potentially reducing the need for prolonged training and resource expenditure.

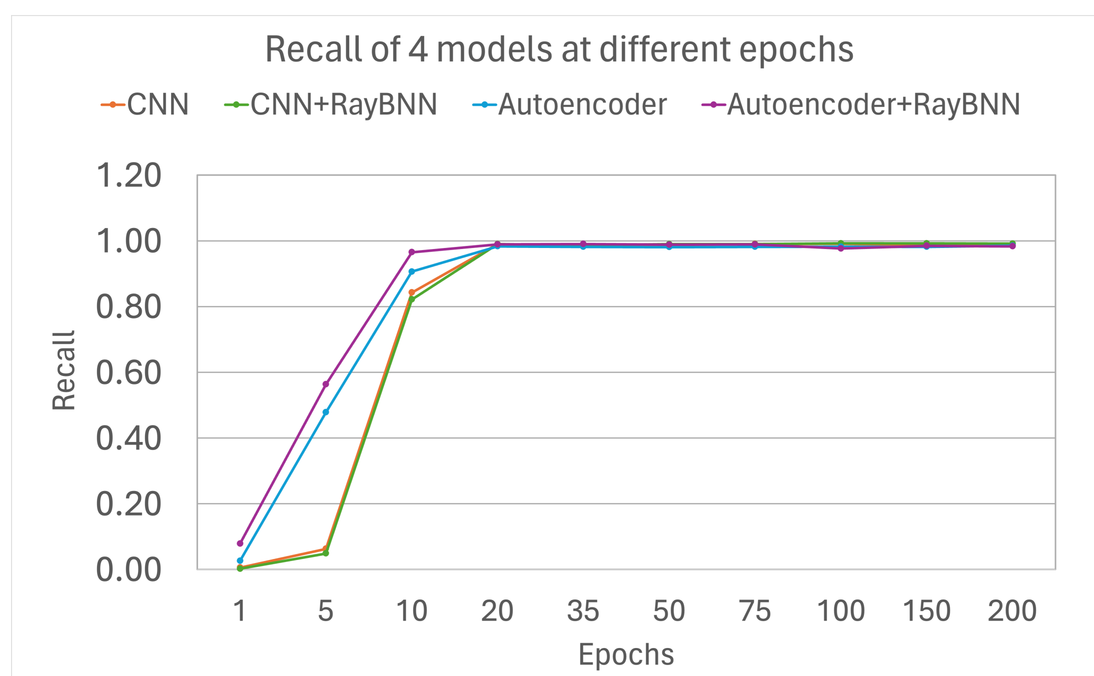


Figure 4.8: Recall of four models at different epochs.

In Fig. 4.8, for the recall metric, the CNN model starts at 0.0058, rising to 0.8425 by epoch 10, which is consistent with precision, ending at 0.9825 by epoch 200, showing the model's strength in identifying all relevant cases. CNN+RayBNN begins at 0.0022, achieving 0.8215 recall by epoch 10, which is slightly lower than precision at this stage, ending at a high of 0.9912 by epoch 200, indicating it becomes increasingly better at identifying true positives. The Autoencoder starts at 0.0265 recall, increasing to 0.9060 by epoch 10, and peaks at 0.9852 by epoch 200. This high recall means it rarely misses out on identifying true positives. The Autoencoder+RayBNN model starts at 0.0789 recall, one of the highest initial recalls, reaching 0.9650 by epoch 10, and ends at 0.9831 by epoch 200, showing a strong and consistent ability to identify true positives.

The Autoencoder model follows a more gradual path to stability, not plateauing until after epoch 50, and it is by epoch 150 that we see it operating at its full potential. Despite the slower initial improvement compared to the CNN models, it demonstrates a steady increase over time, reaching its prime by epoch 150. This suggests that although slower to train, the autoencoder model can still achieve a high level of performance and may benefit from a more extended training period.

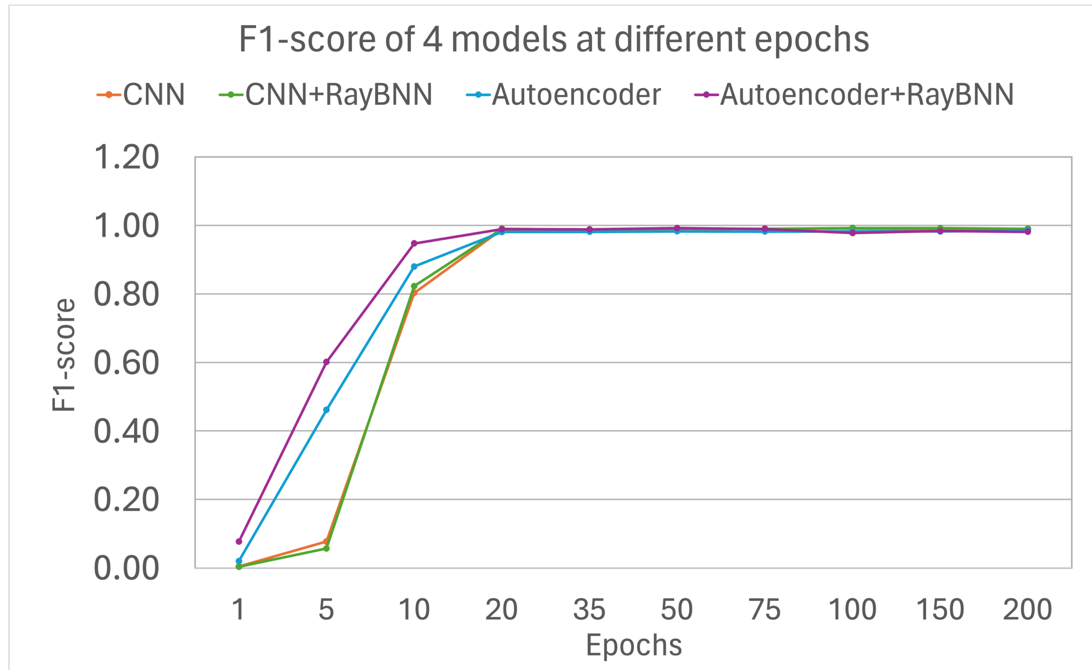


Figure 4.9: F1-score of four models at different epochs.

In Fig. 4.9, the F1-score for the CNN model starts at 0.0042, growing to 0.8026 by epoch 10, reflecting a balance between precision and recall, and converges to 0.9825 by epoch 200, indicating strong overall performance. For CNN+RayBNN, the F1-score commences at 0.0035, reaching 0.8225 by epoch 10, suggesting a balanced precision and recall from an early stage, and ending at 0.9899 by epoch 200, showing the model's robustness. The Autoencoder begins at 0.0198 F1-score, climbing to 0.9800 by epoch 10, and ending at 0.9857 by epoch 200, slightly lower than the CNN models but still indicative of strong predictive performance. Autoencoder+RayBNN starts at 0.0768 F1-score, jumping to 0.9470 by epoch 10, and ends at 0.9809 by epoch 200. The initial high F1-score suggests a strong balance between precision and recall, which is slightly less optimized by epoch 200 compared to other models.

Meanwhile, the Autoencoder+RayBNN model quickly reaches its performance

stride, with metrics stabilizing as early as epoch 20, and achieving the best overall performance by epoch 50. The quick stabilization of metrics could suggest an efficient learning process. However, it also raises concerns regarding the model's potential for overfitting, as indicated by the slight decrease in some metrics beyond this epoch. Therefore, the model may benefit from methods to enhance generalization, such as incorporating regularization techniques or expanding the training dataset.

Analyzing the model performance at various epochs allows us to identify the trends and potential areas for improvement. We observe that CNN and CNN+RayBNN models exhibit relatively stable accuracy, precision, recall, and F1-score throughout the training process. They have minor fluctuations that do not deviate significantly from the overall trend. This stability might imply that these models are capturing the essential patterns within the MNIST dataset effectively and are not suffering from overfitting or underfitting issues. It is especially notable that the addition of RayBNN to the CNN model does not influence the performance metrics but rather maintains a consistent high level of accuracy and precision.

Conversely, Autoencoder and Autoencoder+RayBNN models display more pronounced variability, particularly a marked decrease in performance metrics. Such a trend could be indicative of a model struggling with a specific aspect of the learning process at this stage. For instance, it may suggest that the Autoencoder reaches a point where its representation of the data is overly fitted to the noise within the training set, thereby diminishing its generalization capability.

Upon considering all metrics comprehensively, the CNN+RayBNN emerges as a particularly strong model, striking an effective balance between learning speed and performance. While peak performance is critical, the consistency of performance across epochs is equally valuable, especially in practical applications where stability and reliability over time are paramount.

To enhance models that show early peaking followed by stagnation or decline, several strategies could be employed. Implementing regularization techniques, adding more data to training dataset, or using techniques like early stopping could prevent overfitting. Additionally, adjusting learning rates dynamically during training could help the models to converge more smoothly and potentially reach higher peaks in performance metrics.

Chapter 5

MWT dataset

This chapter only applies CNN+RayBNN to MWT dataset as its performance is slightly better than the autoencoder according to results in the previous chapter. In MWT dataset, one of the 76 recordings, uXdB, contains a large amount of false positive MSE [19]. In addition, in the dataset, 46 recordings contain all types of classifications, while the remaining 30 recordings do not have MSE [37]. This impedes the classification accuracy. To overcome this issue, we combined all recordings in the training and validation datasets. Further, both of the EEG signals in each record are similar, which may cause overfitting. To solve this problem, the dataset is preprocessed by data augmentation using Gaussian noise.

For this dataset, an 7-fold testing scheme is adopted. We split the whole dataset into 7 sections. During each iteration, we kept 5 sections for training, 1 sections for cross-validation, and the remaining section testing dataset. This process is repeated for 7 times such that the average and standard deviation of Cohen's Kappa values are used for model evaluation.

5.1 Hyper-parameter tuning

In the charts we present following, the vertical error bars on each data point represent one standard deviation (σ) of the Cohen's Kappa values.

In this study, we adopted 12 CNN blocks. The first CNN block contains 32 filters, and the 2nd CNN block has 64, the 3rd to 6th CNN blocks all have 128 filters, and the 7th to 12th contain 256 filters. It is proven in deep neural networks such as ResNet [14] and Inception [38], Residual blocks and inception modules being repeated multiple

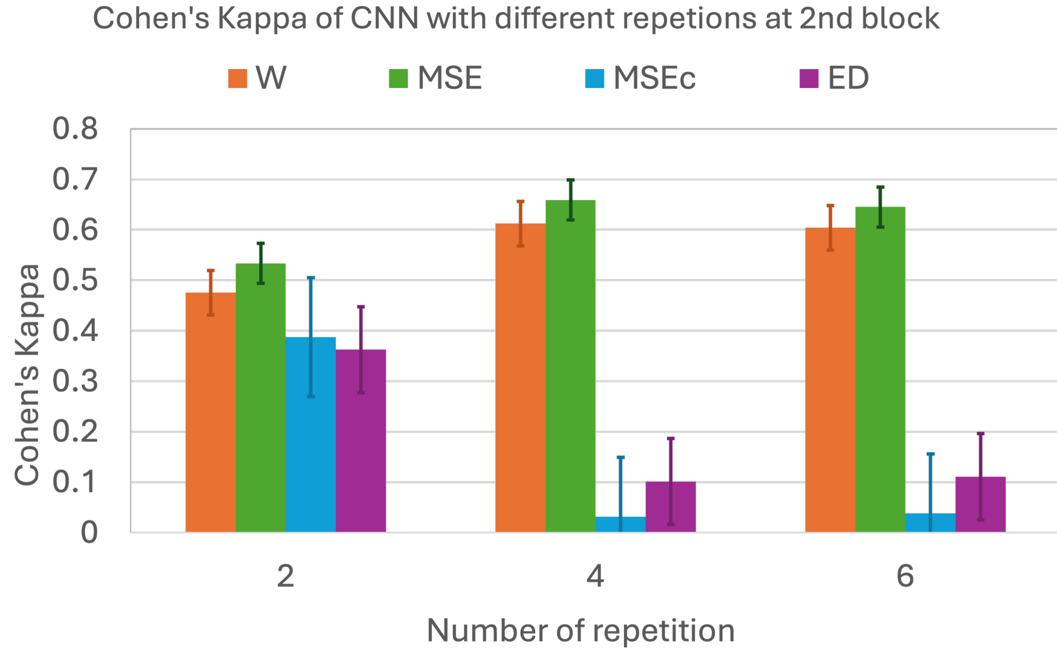


Figure 5.1: Cohen's Kappa of CNN+ANN with different repetitions at 2nd blocks.

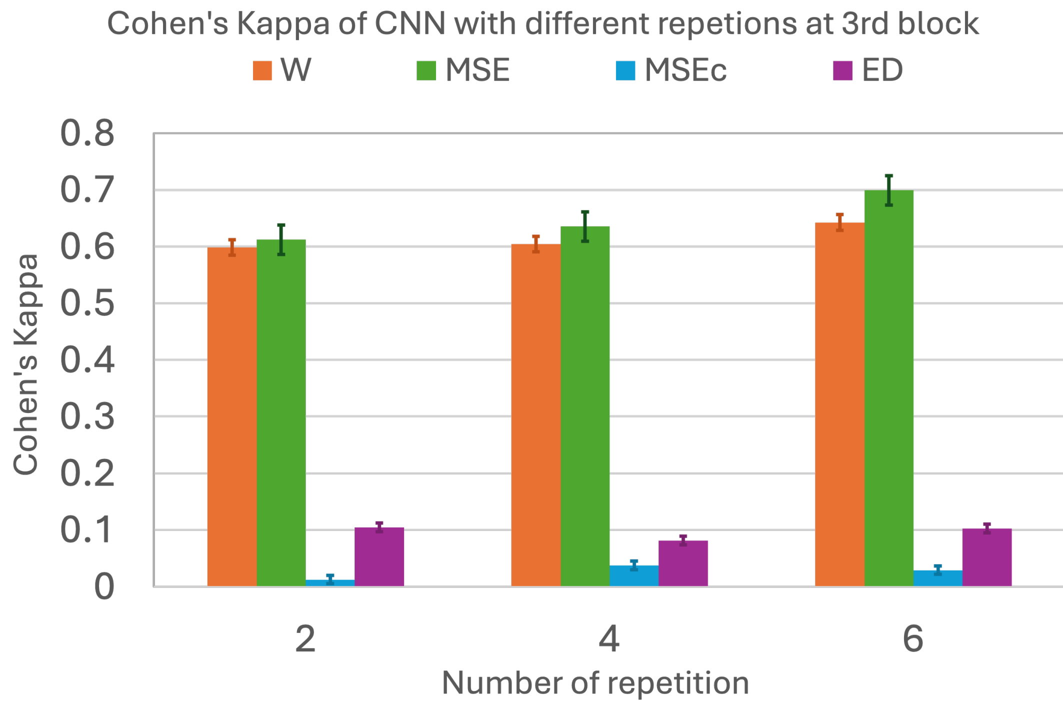


Figure 5.2: Cohen's Kappa of CNN+ANN with different repetitions at 3rd blocks.

times within the network can achieve better performance by capturing multi-scale features. Further, in our study, the filters in CNN blocks have a kernel size of 3×1 and a stride of 1. The max-pooling layer uses a 2×1 pooling size for reducing the dimensions of the features. The first few CNN blocks are set to capture low-level features. As moving deeper into the network, the subsequent layers capture more complex patterns and higher-level features.

We tested the CNN block with 128 filters and 256 filters to find an optimal number of repetitions and display the results in Fig. 5.1 and Fig. 5.2. It indicates that the model with 4 repetitions presents a similar performance to the model with 6 repetitions in the CNN block with 128 filters, thereby we chose to repeat 4 times as a trade-off between computational expense and time-saving. We then applied the same process to the CNN block with 256 filters. It shows that repeating for 6 times at the CNN block performs well, which is the same as that in Ref. [19].

5.2 Results

We first trained the CNN with the ANN model and evaluated its performance on both training and validation datasets. The Kappa values for W class, as shown in Fig. 5.3.

For the training dataset (purple colored dots with solid line), the Kappa value as shown in Fig. 5.3 for W detection, starts higher at 0.66 ± 0.05 in epoch 1. This higher initial value compared to the validation set suggests that the model performs better on the training data, possibly due to overfitting. By epoch 2, the Kappa value decreases to 0.65 ± 0.04, and it further reduces to 0.64 ± 0.05 in epoch 3. This declining trend may indicate that the model is not generalizing as well. The MSE as shown in Fig. 5.4 for the training dataset shows an increasing trend: starting at 0.72 ± 0.05 in epoch 1, rising to 0.74 ± 0.06 in epoch 2, and then slightly decreasing to 0.73 ± 0.06 in epoch 3. This increase in MSE could be an indication that while the model is fitting the training data better initially, it might be struggling to maintain this performance as it continues training. The MSEc as shown in Fig. 5.5 values in the training set remain fairly consistent, with 0.05 ± 0.02 in epoch 1, and maintaining at 0.05 ± 0.02 to epoch 3. The ED as shown in Fig. 5.6 in the training dataset also shows a maintenance, at 0.07 ± 0.02 in 3 epochs. This pattern aligns with the MSE trend, indicating that while there is some variability in the error distance, the model maintains a relatively stable performance.

In the validation dataset (green colored dots with solid line), the Kappa value for detecting W starts at 0.62 ± 0.05 in epoch 1 as shown in Fig. 5.3. This indicates a moderate level of agreement between the model's predictions and the true labels. By epoch 2, the Kappa value improves slightly to 0.63 ± 0.05, suggesting that the model has increased its ability to correctly identify W. However, in epoch 3, the Kappa value decreases to 0.61 ± 0.05, indicating a slight decline in performance after the initial improvement. The MSE as shown in Fig. 5.4 for the validation dataset shows a fluctuating trend: starting at 0.69 ± 0.05 in epoch 1, increasing to 0.70 ± 0.04 in epoch 2, and then decreasing slightly to 0.69 ± 0.05 in epoch 3. This pattern suggests that while the model initially adjusts to the validation data, it experiences some inconsistencies in prediction accuracy. The MSEc as shown in Fig. 5.5 values are relatively stable across epochs, starting at 0.05 ± 0.01 in epoch 1, and further reducing to 0.04 ± 0.02 in epoch 3. This consistent decline implies that the model is progressively refining its error correction mechanisms. The ED as shown in Fig. 5.6 also follows a remaining trend, 0.07 ± 0.02 in 3 epochs. This indicates that the overall distance between the predicted and true values is gradually reducing, reflecting an improvement in the model's precision.

Overall, the model shows a quick learning phase by epoch 2, as indicated by the peak Kappa values in both training and validation datasets. However, the slight decrease in Kappa values by epoch 3, along with the fluctuating MSE and ED, suggests that the model may be starting to early stop on the validation set. These observations highlight the importance of monitoring the model's performance across multiple metrics to ensure it generalizes well to unseen data.

In aggregating the performance across all classifications across both training and validation, we can infer that the optimal point for the model occurs after epoch 2. Therefore, in the current configuration, we adopt early stopping at this epoch to avoid overfitting. Based on the result of CNN at epoch 2, we then extracted the features from CNN and applied them to the RayBNN model. The results are shown in the following analysis.

For the training dataset (orange colored dots with solid line), the Kappa value for W detection as shown in Fig. 5.3 starts at 0.18 ± 0.03 in epoch 1, indicating a better initial fit to the training data. By epoch 10, the Kappa value improves to 0.26 ± 0.04, and it significantly increases to 0.46 ± 0.04 by epoch 35. The peak Kappa value of 0.68 ± 0.06 is observed at epoch 50, indicating the model's optimal performance. By epoch 75, the Kappa value increases slightly to 0.69 ± 0.04, and by epoch 100, it

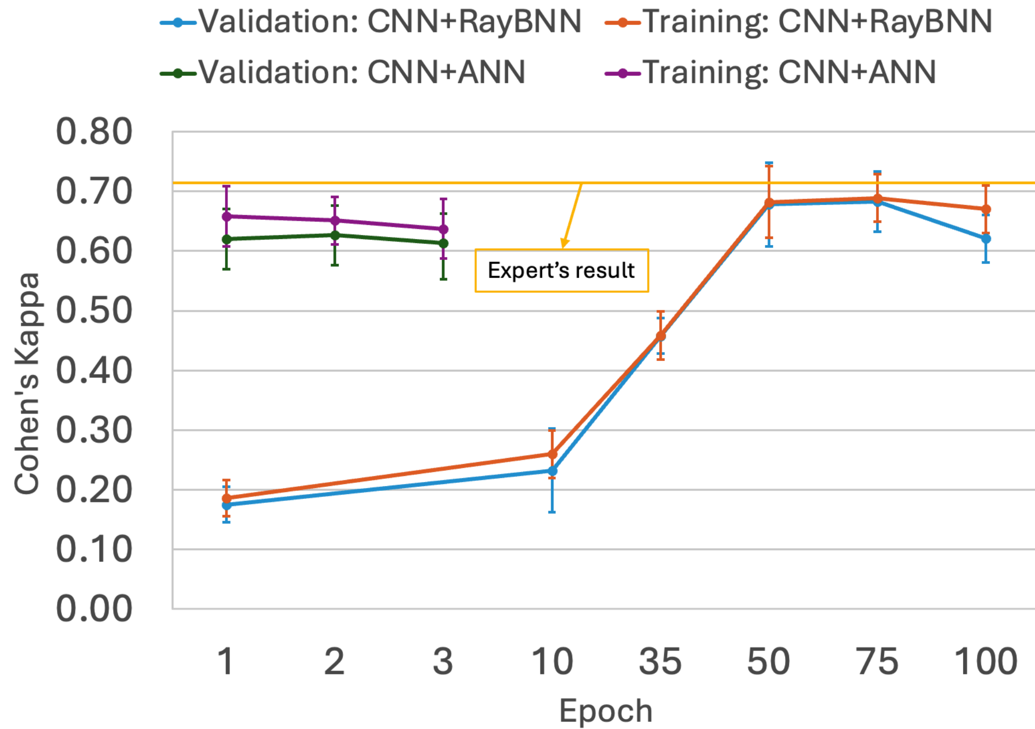


Figure 5.3: Kappa values of CNN+ANN and CNN+ANN with RayBNN models for W classification at different epochs. The yellow line indicates the Cohen's Kappa value from experts.

decreases slightly to 0.67 ± 0.04, suggesting some fluctuation in performance. The MSE as shown in Fig. 5.4 for the training dataset starts at 0.24 ± 0.07 in epoch 1, indicating the initial error rate. This value increases to 0.36 ± 0.05 by epoch 10 and continues to rise, reaching 0.56 ± 0.09 by epoch 35. The highest MSE of 0.77 ± 0.05 is observed at epoch 50. The MSE remains at 0.77 ± 0.04 at epoch 75, and by epoch 100, it slightly decreases to 0.76 ± 0.07, indicating a minor improvement in error rate. The MSEc as shown in Fig. 5.5 values in the training set start extremely low at almost 0.00 ± 0.01 in epoch 1. By epoch 10, MSEc remains at 0.00 ± 0.01, and it increases to 0.03 ± 0.02 by epoch 35. The highest MSEc value of 0.04 ± 0.02 is observed at epoch 50 and remains the same at epoch 75. By epoch 100, MSEc decreases slightly to 0.03 ± 0.01, indicating some error correction stabilization. The ED as shown in Fig. 5.6 in the training dataset starts very low at 0.00 ± 0.01 in epoch 1 and increases to 0.02 ± 0.01 by epoch 10. It continues to rise, reaching 0.03 ± 0.01 by epoch 35 and peaking at 0.05 ± 0.02 by epoch 50. The ED remains at 0.05 ± 0.01 at epoch 75 and

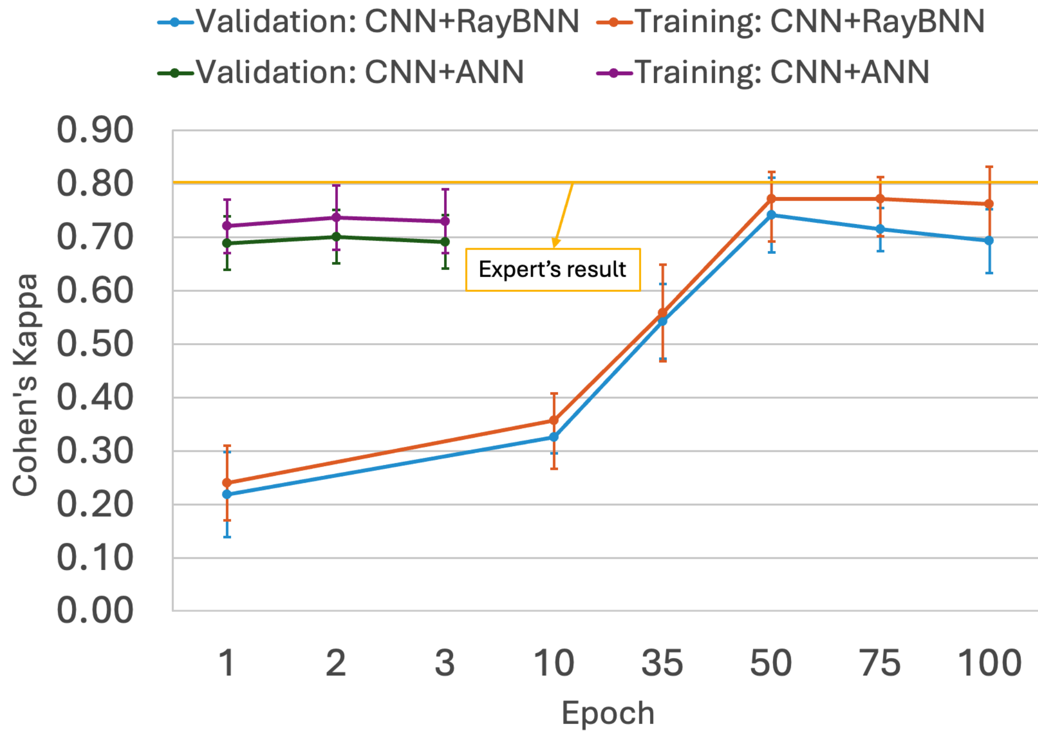


Figure 5.4: Kappa values of CNN+ANN and CNN+ANN with RayBNN models for MSE classification at different epochs. The yellow line indicates the Cohen's Kappa value from experts.

increases slightly to 0.07 ± 0.02 by epoch 100.

In the validation dataset (blue colored dots with solid line), the Kappa value for detecting W as shown in Fig. 5.3 starts at 0.18 ± 0.03 in epoch 1. This initial value indicates a relatively low agreement between the model's predictions and the true labels. By epoch 10, the Kappa value improves to 0.23 ± 0.07, showing some early learning. A significant increase is observed by epoch 35, with the Kappa value rising to 0.46 ± 0.03, and it continues to improve, reaching 0.68 ± 0.07 by epoch 50. This suggests that the model becomes considerably more accurate in detecting W as training progresses. The Kappa value remains at 0.68 ± 0.05 at epoch 75, indicating stable performance, but it slightly decreases to 0.62 ± 0.04 by epoch 100, indicating some fluctuation in the model's performance. The MSE as shown in Fig. 5.4 for the validation dataset begins at 0.22 ± 0.08 in epoch 1, indicating initial prediction errors. This value increases to 0.33 ± 0.03 by epoch 10 and further to 0.54 ± 0.07 by epoch 35, reflecting the complexity of the task. The highest MSE is observed at

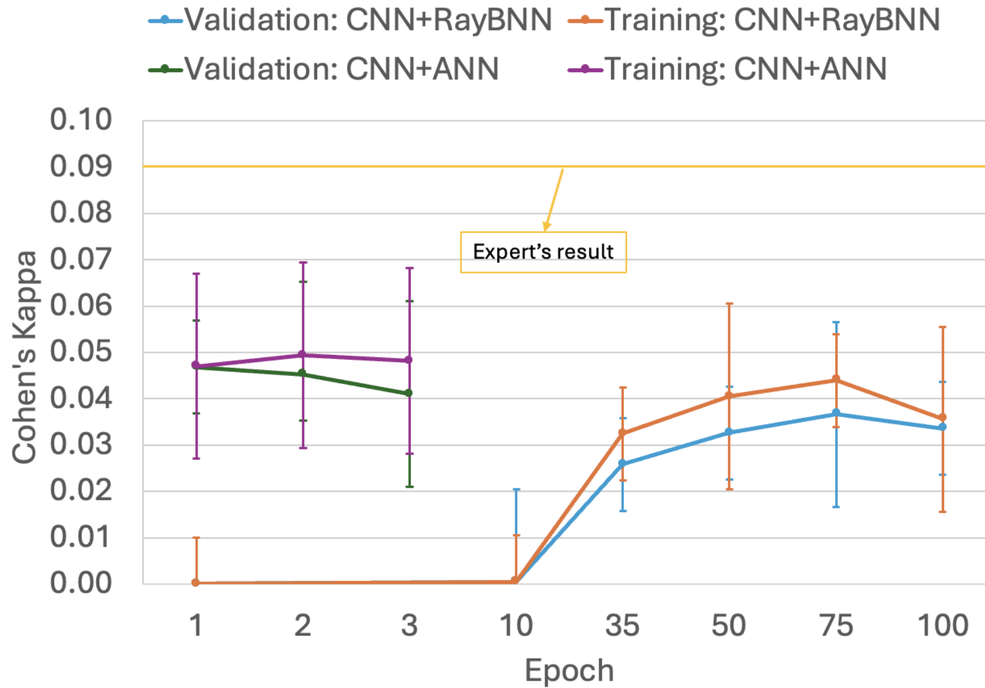


Figure 5.5: Kappa values of CNN+ANN and CNN+RayBNN models for MSEc classification at different epochs.The yellow line indicates the Cohen's Kappa value from experts.

epoch 50, with a value of 0.74 ± 0.07, suggesting that the model encounters some difficulties in maintaining low error rates. By epoch 75, the MSE slightly decreases to 0.71 ± 0.04, and by epoch 100, it further reduces to 0.69 ± 0.06, indicating some improvement in prediction accuracy. The MSEc as shown in Fig. 5.5 values for the validation dataset start extremely low at almost 0.00 ± 0.01 in epoch 1, reflecting minimal initial correction. By epoch 10, MSEc remains at almost 0.00 ± 0.02, and it starts to rise, reaching 0.03 ± 0.01 by epoch 35 and 0.04 ± 0.01 by epoch 50. This trend indicates the model's increasing capacity to correct errors over time. The ED starts very low at almost 0.00 ± 0.01 in epoch 1 and increasing to 0.02 ± 0.01 by epoch 10. The ED as shown in Fig. 5.6 reaches 0.03 ± 0.02 by epoch 35 and 0.06 ± 0.01 by epoch 50, reflecting the increasing distance between predicted and true values as the model continues to learn. By epoch 75, ED remains at 0.06 ± 0.02, and by epoch 100, it rises slightly to 0.07 ± 0.02, indicating some variability in the model's precision.

The human experts' scores of the MWT dataset are mentioned in Char. 3, which

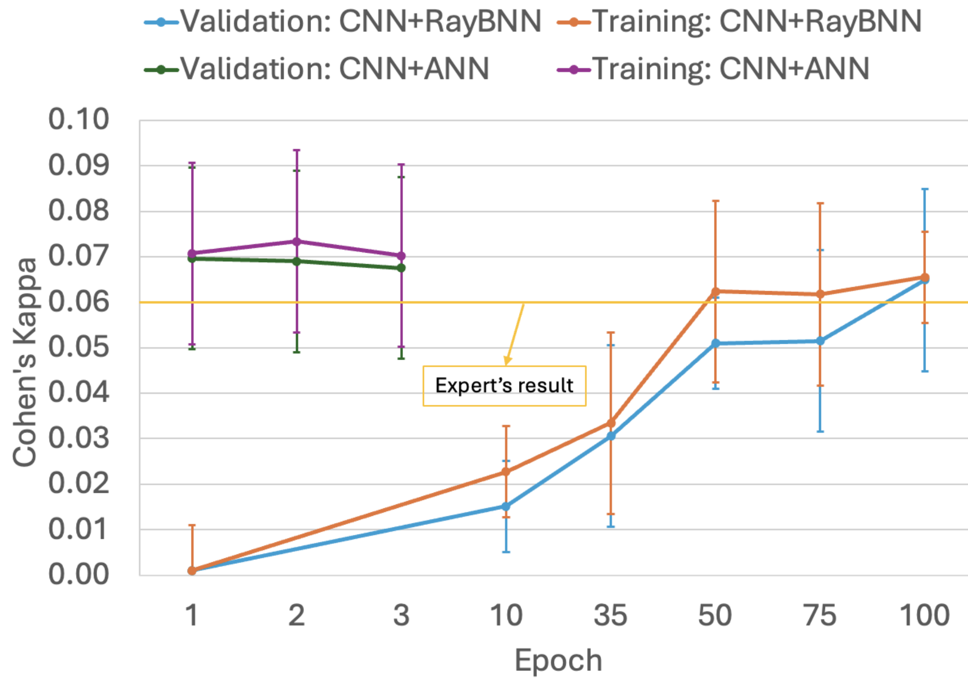


Figure 5.6: Kappa values of CNN+ANN and CNN+RayBNN models for ED classification at different epochs.The yellow line indicates the Cohen's Kappa value from experts.

are used to evaluate the accurate identification of sleep stages. Comparing the results of CNN+ANN model and CNN+RayBNN model in Fig. 5.3, Fig. 5.4, Fig. 5.5, and Fig. 5.6, with the expert scores-0.71 for W, 0.80 for MSE, 0.09 for MSEc, and 0.06 for ED (yellow solid line) in the charts, the model comes close to expert-level performance for W and MSE classes, while in MSEc and ED states, the results present lacks in performance. By epoch 75, the CNN+RayBNN model presents a good performance in Kappa values for W, MSE, MSEc, and ED classes, where it achieves 0.68 ± 0.05, 0.71 ± 0.04, 0.04 ± 0.02, and 0.06 ± 0.02 respectively in the validation set. The lacks in MSEc and ED classes present the inherent difficulty in classifying these ambiguous states, which often do not exhibit clear, distinguishable patterns. The low proportion of MSEc and ED states in this dataset should also be considered when analyzing the situation of low kappa values.

We evaluated the models on the testing sets that we saved at the beginning of splitting the dataset. The CNN+ANN with RayBNN resulted in Cohen's Kappa

values as 0.62 ± 0.02 for W, 0.70 ± 0.02 for MSE, 0.05 ± 0.01 for MSEc, and 0.08 ± 0.02 for ED, which is close to the results that we evaluated in validation sets. The Cohen's Kappa values of CNN+ANN model are 0.54 ± 0.09 for W, 0.63 ± 0.14 for MSE, 0.04 ± 0.03 for MSEc, and 0.02 ± 0.02 for ED, which is also close to the results in the validation dataset. Both models are close to expert's results in Cohen's Kappa metric, but the CNN+ANN with RayBNN is slightly better than CNN+ANN approach in W and MSE detections. For MSEc and ED detections, both models are restricted to the ambiguous features, and they still have room to improve.

We also calculated the computation time for each training epoch to evaluate the efficiency and the practicality of both CNN+ANN and CNN+RayBNN models. The CNN+ANN model exhibited an average computation time of 4673 seconds per epoch. This duration can be attributed to the complex structure of the CNN with ANN, which involves a large number of parameters and dense connections, leading to increased computational time. In contrast, the CNN+RayBNN model demonstrated a markedly lower average computation time of 443 seconds per epoch. The RayBNN leverages random neuron connections and optimized data processing pathways. This efficiency not only accelerates the training process but also highlights the potential of RayBNN in handling large-scale and dynamic datasets more effectively than traditional CNN+ANN model. The significant disparity in computation times underscores the advantages of integrating RayBNN with CNN, offering a more time-efficient solution for neural network training.

Chapter 6

Conclusions

In this study, we present approaches that utilize the strengths of Convolutional Neural Networks with Artificial Neural Networks and Autoencoder and apply them to ray-traced Biological Neural Networks to address the limitations of traditional neural network architectures. By integrating the feature extraction capabilities of CNNs with ANNs and autoencoder, and the adaptability of RayBNN, the hybrid models demonstrate versatility and efficiency in processing and adapting to various data types and formats.

The usage of hybrid models that can dynamically reconfigure themselves in response to new datasets. This flexibility is significantly enhanced by combining RayBNN with CNN+ANN and CNN+Autoencoder approaches. We proved that the CNN with ANN approach performed well in extracting features from the MNIST dataset and MWT dataset, ensuring that the RayBNN receives the most relevant information. Because of the special characteristics of RayBNN-the neurons in it are related to each other randomly, and there is no specific concept of layer, thereby enabling a more efficient information transferring flow. Though the performance of the CNN+Autoencoder approach with RayBNN is not as effective as the CNN+ANN approach with RayBNN, it still obtains high-level results in various evaluation metrics using its professional ability of dimension reduction and feature extraction.

The result of CNN+ANN with RayBNN structure proved that this model can recognize wakefulness and microsleep episodes well, which is close to the result of a human expert. Currently, the model is only trained on one image dataset and one EEG dataset, more and larger datasets are needed to achieve an overall assessment.

Future research would focus on further refining this model and exploring its potential applications across various industries, including autonomous systems, health

diagnostics, and financial forecasting. The innovative approach outlined in this paper underscores the potential of combining CNN with different approaches and apply to RayBNN to revolutionize the way traditional neural networks are designed and utilized, providing more adaptable and efficient solutions in the realm of artificial intelligence.

In addition, the model can be further enhanced by training CNN and RayBNN together. In this way, different structures are not only connected by extracted feature data but also by transferring gradients and learning signals from RayBNN for further modification by CNN. Such an approach potentially enhances the learning process of CNN. This could lead to better feature extraction and improved overall performance of the combined model. Besides, training models together allow CNN to adapt itself dynamically based on the feedback from RayBNN. This can help the CNN to adjust its parameters more effectively in response to changing data or tasks, which can lead to more efficient backpropagation of errors through models.

Bibliography

[1] Nidhi Agarwal, Akanksha Sondhi, Khyati Chopra, and Ghanapriya Singh. Transfer learning: Survey and classification. In *Advances in Intelligent Systems and Computing*, 2021.

[2] Bruce Alberts. *Essential Cell Biology(3rd edition)*. Garland Science, New York, 2009.

[3] N. I. Chervyakov, P. A. Lyakhov, M. A. Deryabin, N. N. Nagornov, M. V. Valueva, and G. V. Valuev. Residue number system-based solution for reducing the hardware cost of a convolutional neural network. *Neurocomputing*, 407:439–453, September 2020.

[4] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 1960.

[5] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[6] Karl Doghramji, Merrill M Mitler, R B Sangal, Colin Shapiro, Susan Taylor, Joyce Walsleben, Charles Belisle, Milton K Erman, Robert Hayduk, Rose Hosn, Edward B O'Malley, Joseph M Sangal, Steven L Schutte, and Joseph M Youakim. A normative study of the maintenance of wakefulness test. *Electroencephalography and Clinical Neurophysiology*, 103(5):554–562, November 1997.

[7] B. Fasel. An introduction to bio-inspired artificial neural network architectures. *Acta Neurologica Belgica*, 103(1):6–12, 2003.

[8] Yang Gao. Complex-wavelet structural similarity based image classification, 2012.

[9] Jacinta Goddard, Georgina Tay, Jessica Fry, Melissa Davis, Darcy Curtin, and Ildiko Szollosi. Multiple sleep latency test: when are 4 naps enough? *Journal of Clinical Sleep Medicine*, 17(3):491–497, March 2021.

[10] Patrick J. Grother. Nist special database 19 - handprinted forms and characters database, 1995.

[11] C. Hara, F. L. Rocha, and M. F. F. Lima-Costa. Prevalence of excessive daytime sleepiness and associated factors in a brazilian community: the bambuí study. *Sleep Medicine*, 5:31–36, 2004.

[12] L. Hardesty. Explained: Neural networks. MIT News, April 2017.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, 1512.03385, 2015.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, June 2016.

[15] Anne Hertig-Godeschalk, Joscha Skorucak, Alexander Malafeev, Peter Achermann, Johannes Mathis, and David R. Schreier. Microsleep episodes in the borderland between wakefulness and sleep. *Sleep*, 43:163, 2020.

[16] Sukumar Katamreddy, Pat Doody, Joseph Walsh, and Daniel Riordan. Visual udder detection with deep neural networks. In *2018 12th International Conference on Sensing Technology (ICST)*, 2018.

[17] G. H. Klem et al. The ten-twenty electrode system of the international federation. *Electroencephalography and Clinical Neurophysiology*, 52:3–6, 1999.

[18] H. Liang, W. Fu, and F. Yi. A survey of recent advances in transfer learning. *2019 IEEE 19th International Conference on Communication Technology (ICCT)*, 2019.

[19] Alexandr Malafeev, Anne Hertig-Godeschalk, D. R. Schreier, Jelena Skorucak, Johannes Mathis, and Peter Achermann. Automatic detection of microsleep episodes with deep learning. *Frontiers in Neuroscience*, 15:564098, 2021.

[20] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of Learning and Motivation*, 24:109–165, 1989.

[21] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[22] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[23] C.T. Nguyen, N. Van Huynh, N.H. Chu, Y.M. Saputra, D.T. Hoang, D.N. Nguyen, Q.-V. Pham, D. Niyato, E. Dutkiewicz, and W.-J. Hwang. Transfer learning for wireless networks: A comprehensive survey. *Proceedings of the IEEE*, 2022.

[24] S. Niu, Y. Liu, J. Wang, and H. Song. A decade survey of transfer learning (2010–2020). *IEEE Transactions on Artificial Intelligence*, 1(2):151–166, 2020.

[25] B Oxkey. International 10-20 system for eeg electrode placement, showing modified combinatorial nomenclature, 2017.

[26] Amit Paul, Linda Ng Boyle, Jon Tippin, and Matthew Rizzo. Variability of driving performance during microsleeps. In *Proceedings of the Third International Driving Symposium on Human Factors in Driver Assessment, Training and Vehicle Design*, 2005.

[27] Govinda R Poudel, Carrie R Innes, Philip J Bones, Richard Watts, and Richard D Jones. Losing the struggle to stay awake: Divergent thalamic and cortical activity during microsleeps. *Human Brain Mapping*, 35(1):257, 2012.

[28] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[29] Frank Rosenblatt. The perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

[30] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. *Learning Internal Representations by Error Propagation*. MIT Press, 1986.

[31] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, December 2015.

[32] Dominik Scherer, Andreas C. Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *ICANN (3)*, 2010.

[33] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[34] L. Shao, F. Zhu, and X. Li. Transfer learning for visual categorization: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 26(5):1019–1034, 2014.

[35] G. Shaw. Cpca-gfap, mca-5b10, tau, neurons. `https://commons.wikimedia.org/wiki/File:CPCA-GFAP,_MCA-5B10,_Tau,_neurons.jpg`, 2017.

[36] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[37] Joscha Skorucak, Anne Hertig-Godeschalk, David R Schreier, Alexander Malafeev, Johannes Mathis, and Peter Achermann. Automatic detection of microsleep episodes with feature-based machine learning. *Sleep*, 43(1):225, January 2020.

[38] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[39] Ragav Venkatesan and Baoxin Li. *Convolutional Neural Networks in Visual Computing: A Concise Guide*. CRC Press, 2017.

[40] Jeremy West, Dan Ventura, and Sean Warnick. Spring research presentation: A theoretical foundation for inductive transfer. Brigham Young University, College of Physical and Mathematical Sciences, 2007.

[41] K. W. Yau. Receptive fields, geometry and conduction block of sensory neurones in the central nervous system of the leech. *The Journal of Physiology*, 263(3):513–538, December 1976.

[42] Brosnan Yuen, Xiaodong Dong, and Tao Lu. A 3D ray traced biological neural network learning model. *Nature Communications*, 15:4693, 2024.

[43] Andreas Zell. *Simulation Neuronaler Netze (1st Edition)*. Addison-Wesley, 1994.

[44] Yingjie Zhang, Hong Geok Soon, Dongsen Ye, Jerry Ying Hsi Fuh, and Kunpeng Zhu. Powder-bed fusion process monitoring by machine vision with hybrid convolutional neural networks. *IEEE Transactions on Industrial Informatics*, 16(9):5769–5779, September 2020.