

Photocopy and Use Authorization

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at Idaho State University, I agree that the Library shall make it freely available for inspection. I further state that permission for extensive copying of my thesis for scholarly purposes may be granted by the Dean of Graduate Studies, Dean of academic division, or by the University Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature_____

Date_____

**An Obstacle Classification Method
using Convolutional Neural Network
for Autonomous Wheelchair Navigation**

by

Anupama Shree Dhamala

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in the Department of Measurement and Control Engineering

Idaho State University

Summer 2019

To the Graduate Faculty:

The members of the committee appointed to examine the thesis of ANUPAMA S. DHAMALA find it satisfactory and recommend that it be accepted.

Dr. Marco P. Schoen, Major Advisor

Dr. Kenneth Bosworth, Committee Member

Dr. Steve Chiu, Graduate Faculty Representative

ACKNOWLEDGEMENTS

I would first like to thank my major advisor, Dr. Marco Schoen for his assistance and guidance in completing this thesis. I wish to thank him for his patience and kindness. Without his knowledge, motivation, and support this research would not have been possible. I thank Dr. Steve Chiu and Dr. Alba Perez for supporting me with Career Path Internships and Graduate Teaching Assistantships. Thank you, Dr. Chiu for constantly motivating and encouraging me. Thank you, Dr. Alba for being an inspiring mentor and for believing in me. I would like to express my deep gratitude for Dr. Bosworth, for his willingness to give me time and for teaching me math.

My sincere thanks to the Electrical and Mechanical Engineering Departments of ISU. I would like to express my gratitude for Ellen and Laura for helping me throughout the way.

I thank my classmates in the python class of Summer 2018, which propelled my research further. To everyone in MCERC, thank you for all your encouragement, it was a wonderful experience sharing a lab with you. I thank the librarians of Eli M Oboler library (especially, Laura, Spencer and Dr. Homan) who helped me with research, interlibrary loans and for their constant encouragement. I would like to extend my warm thanks to Debbie from Scholarship office for her understanding and patience, Anna from Graduate school office for all her help and guidance. My immense thanks to the open source community for sharing their learning which has been instrumental in my research.

I thank my friends and family for their support and patience throughout my study. I am especially grateful to Sabin and Naga for mentoring me, for their unwavering support and for their presence in the ERC lab. To Anup, Anurupa, Jo, Kumari, Wilson, Colin, Faith, Kunal, Aayush, Joan, Arati and Sneha, thank you!

TABLE OF CONTENTS

List of Figures	vi
Abstract	viii
1. INTRODUCTION	1
1.1 Background.....	1
1.2 Literature Review	2
1.3 Thesis Goal	3
1.4 Thesis Organization	3
2. THEORY	4
2.1 Machine Learning.....	4
2.2 Artificial Neural Network.....	4
2.2.1 Biological Inspiration.....	4
2.2.2 An Artificial Neuron Model.....	6
2.2.3 Back-propagation	8
2.2.4 Theoretical background of back-propagation	8
2.2.5 Activation functions	11
2.2.6 Cost Functions.....	16
2.2.7 Weight initialization.....	19
2.3 Convolutional Neural Network	20
2.3.1 Convolution Layer.....	20

2.3.2 ReLU layer	24
2.3.3 Pooling	24
2.3.4 Flattening.....	26
2.3.5 Full Connection	26
2.4 Keras	27
2.4.1 Keras Installation.....	27
2.4.2 Basic functions of Keras	28
2.5 Kinect.....	28
2.6 Kinect control using Python	29
2.7 Wheelchair.....	30
3. EXPERIMENTAL SETUP AND IMPLEMENTATION OF NEURAL NETWORK	31
3.1 Experimental Setup.....	31
3.1.1 Block Diagram	31
3.1.2 Current Setup for Image Capture	31
3.1.3 Computer Specification.....	32
3.2 Artificial Neural Network.....	32
3.2.1 Predictive Churn Model	32
3.2.2 Dataset and Data preprocessing	32
3.2.3 ANN implementation using Keras	34
3.2.5 Optimizing and tuning the ANN	37

3.3 Convolutional Neural Network (Image Classification).....	37
3.3.1 Image Classification.....	37
3.3.2 Dataset.....	37
3.3.3 CNN implementation in Keras.....	38
4. RESULT AND DISCUSSION	42
4.1 Output of the CNN	42
4.2 Testing different configurations of CNN.....	42
4.3 Testing with novel images.....	43
4.4 Optimizing the CNN.....	47
5. CONCLUSION AND FUTURE WORK	49
5.1 Conclusion.....	49
5.2 Future Work.....	49
6. REFERENCES	51
7. APPENDIX.....	55
7.1 Data preprocessing for Artificial Neural Network	55
7.2 Image pre-processing for Convolution Neural Network	55
7.3 Single Prediction in Convolutional Neural Network using Keras.....	56
7.4 Image Capture using Kinect	56

LIST OF FIGURES

Figure 2.1: Biological Neural System.....	5
Figure 2.2: Schematic diagram of biological neurons [14].....	6
Figure 2.3: A simple artificial neuron model[15]	7
Figure 2.4: A simple Artificial Neural Network[12]	7
Figure 2.5: Linear activation function[18].....	12
Figure 2.6: Sigmoid activation function[18].....	13
Figure 2.7: Hyperbolic Tangent activation function[18]	14
Figure 2.8: Rectified Linear Unit activation function[18]	15
Figure 2.9: ReLU and Leaky ReLU[18]	16
Figure 2.10: Binary cross entropy function, $Y=0$ [20]	18
Figure 2.11: Binary cross entropy function, $Y=1$ [20]	18
Figure 2.12: Structure of Convolutional Neural Network [23].....	20
Figure 2.13: (a) Lenna Image, (b) Horizontal Edges Highlight, (c) Vertical Edges Highlight [22]	23
Figure 2.14: ReLU function[18]	24
Figure 2.15: Fully Connected Layer[33]	27
Figure 3.1: Block diagram of the experimental setup.....	31
Figure 3.2: Setup for Image Capture.....	31
Figure 3.3: Sample of dataset for churn modeling.....	32
Figure 3.4: Artificial Neural Network Code in Keras.....	34
Figure 3.5: Output of the ANN code	36
Figure 3.6: Image example for (a) Curb and (b) Edge.....	38

Figure 3.7: Building a CNN in Keras	38
Figure 3.8: Training a CNN in Keras.....	39
Figure 4.4: Images not included in the training and test sets (set 1).....	44

ABSTRACT

An obstacle classification method using Convolutional Neural Network for autonomous wheelchair navigation

Thesis Abstract--Idaho State University (2019)

This thesis aims to build a curb and edge detection method for autonomous wheelchair navigation using Convolutional Neural Networks (CNNs). In contrary to a deterministic approach, CNNs use a statistical inference approach to the classification problem. For this, a comprehensive dataset of curb and edge examples is collected. The training and test dataset was collected around Idaho State University and the Measurement and Control Engineering Research Center. After the neural network is trained using the dataset, it is tested using images not utilized for training.

The entire system consists of the electric wheelchair, a Kinect 360 and a laptop. The Kinect 360 is used to capture images, and the laptop runs the CNN algorithm to classify the collected pictures. The configuration of the CNN is such that a number of feature detectors and a number of hidden layer neurons are varied in order to test the accuracy of the classification algorithm. An optimum configuration for the available training and test set is realized.

The final CNN shows almost 100 percent accuracy for images with distinct curbs and edges. It still shows reasonable accuracy for images with partially hidden curbs or gently sloping edges.

Key Words: Obstacle Detection, Vision Processing, Autonomous Vehicles, Autonomous Wheelchair,

Convolutional Neural Networks, Kinect 360, Curbs and Edges Detection, Deep Learning, Keras, OpenNI, OpenCV

1. INTRODUCTION

Neural Networks (NN) were first conceived in the late 1950s, [1]. There were two primary reasons behind why neural networks couldn't be implemented to a large extent when they were first developed. The two reasons were computational power and data/memory. The computational power during that time was simply not enough for training neural networks. The advent of Graphic Processing Units was crucial to the implementation and subsequent rise of NN. The second reason is the lack of data. Neural network algorithms are very data hungry. Without enough data, overfitting occurs. The real breakthrough for neural networks happened in 2012 when a Convolutional Neural Network (CNN) called AlexNet, designed by Alex Krizhevsky, won the ImageNet Large-Scale Visual Recognition Challenge, [2]. Every year since then, the challenge has been won by CNNs, [3]. Due to its proven track record in image classification problems, CNN can be considered as one of the best ways for detection of obstacles such as curbs and edges in autonomous vehicles.

1.1 Background

In USA, there were 3.3 million people using wheelchair in 2011, [4]. According to Wheelchair Foundation, with the world population increasing by 187,500 each day, there is an additional need for almost 3,500 wheelchairs every day (1.85% of the world population). That means that every hour of every day, there is the need for 145 more wheelchairs in the world. This statistic is based on a study conducted in 2016, [5]. A number of wheelchair users utilize powered wheelchairs due to the severity of their disability. The power wheelchair market was expected to be \$3.9 billion in 2018 which translates to around 550,000 wheelchairs, [6],[30].

A research by the American Congress of Rehabilitation Medicine in 2011 shows that tips and falls accounted for 87.8% of accidents in motorized wheelchairs. Of this percentage, 20-30% of the

accidents were caused by uneven surfaces, [7].

1.2 Literature Review

Many companies and institutions including DARPA, Google and Stanford university, among others, are currently working on autonomous vehicles technology, [8]. Among the many problems that need to be considered to develop an autonomous vehicle for an urban environment, planning, control and perception of the vehicle are the most important. The autonomous vehicle needs to plan its route, control its speed and direction to follow the planned route and finally be able to perceive its surrounding to avoid any collisions. Without any one of these, the vehicle will fail in its desired function.

Intelligent wheelchair solutions like NAVchair, [9] have been developed for elderly people who use ultrasonic and infrared sensors for obstacle detection. Drive-Safe Systems (DSS) have also been implemented for visually impaired users, [10]. DSS allows users to safely navigate by following walls. It can identify obstacles using two bumpers, five infrared and five ultrasonic sensors. While these sensors based navigation systems are simple and easy to install, they can run into issues like reflections and poor angular resolution. It collects data from the sensor nodes from around the wheelchair and can detect obstacles as low as three inches and as high as 60 inches from ground level. If the DSS detects the obstacle it can slow down or even stop the wheelchair to avoid collision, [10].

Vision-based perception can be considered as an alternative to sensor-based perception. These methods can use depth information as well as image processing and computer vision based solutions to detect obstacles and navigate surroundings, [11].

1.3 Thesis Goal

The goal of the thesis can be summarized as stated below:

- gain an understanding of Artificial and Convolutional Neural Networks (CNNs)
- interface between the Kinect sensor and Python code to capture images for classification by the use of neural network
- collect a dataset of curbs and edges for training and test of CNN
- implement curb and edge classifier using CNN

1.4 Thesis Organization

In Chapter 2, Artificial Neural Network (ANN), Convolutional Neural network (CNN), Kinect, and programming tools like Python, Keras, Open Natural Interaction and Open Computer Vision libraries are discussed. Chapter 3 focuses on implementation of ANN and CNN in Python using Keras, dataset used for training the two neural networks and the output of the neural network. In Chapter 4, the experimental setup for image capture and classification is described. Chapter 5 discusses the conclusion of the research and future work needed to move further towards the end goal. The appendix contains the code that was used in the research but weren't included in Chapter 3.

2. THEORY

2.1 Machine Learning

Machine learning is training machines using algorithms and statistics. It is completing tasks through experience and pattern recognition, as opposed to following hard and fast rules. It does so by analyzing data, identifying patterns, and making decisions with little to no human intervention.

2.2 Artificial Neural Network

Artificial Neural Networks (ANNs) are one of the most important techniques used in machine learning, [12]. As the name suggests, it consists of a network of nodes with each node acting as a single neuron. It tries to emulate human brain in this manner. Thus, they are bio-inspired and are intended to learn as humans do. They recognize patterns in numerical data, normally organized in clusters or vectors, into which all real-world data must be translated. For example, images can be translated into their RGB values, customers can be translated into their specifics like age, weight, income, etc.

2.2.1 Biological Inspiration

A human neural system consists of three stages: receptors, neural network and effectors. In Figure 2.1, the receptors receive signals in form of stimuli from either external or internal sources and send them to the neural network in form of electrical impulses. The neural network processes the electrical signal and provides the proper decision as output in form of electrical impulses. The effectors finally translate the electrical impulses into proper response to the outside environment. The communication is not unidirectional; there is a feedback loop between all three stages, [13].

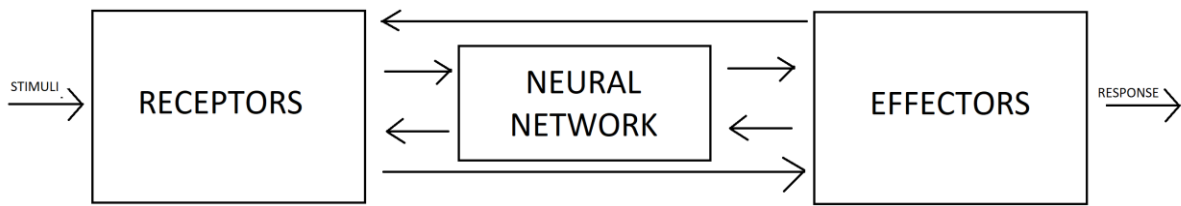


Figure 2.1: Biological Neural System

The basic component of the neural network is a neuron. A neuron mainly has three distinct parts, dendrites, cell body and axon, as shown in Figure 2.2. Dendrites are tree-like structures that receive signals from other neurons where each line is connected to a single neuron. The cell body takes in signals from the dendrites, then sums and thresholds them. The axon carries the signal from cell body to other neurons. The axon connects to dendrites through synapses. The signal transfer through the synapses is usually chemical diffusion but may be electrical impulses, [13].

The signals from each synapse may excite or inhibit the target neuron i.e., helping or hindering firing. The neuron fires when certain conditions are met such as, excitation signals exceeding inhibitory signal by a certain amount in a given period of time. If weight is assigned to each incoming signal, the excitation signals have positive weight while inhibitory signals have negative weight. Therefore, it can be said that: “A neuron fires only if the total weight of the synapses that receive impulses in the period of latent summation exceeds the threshold.”, [13]

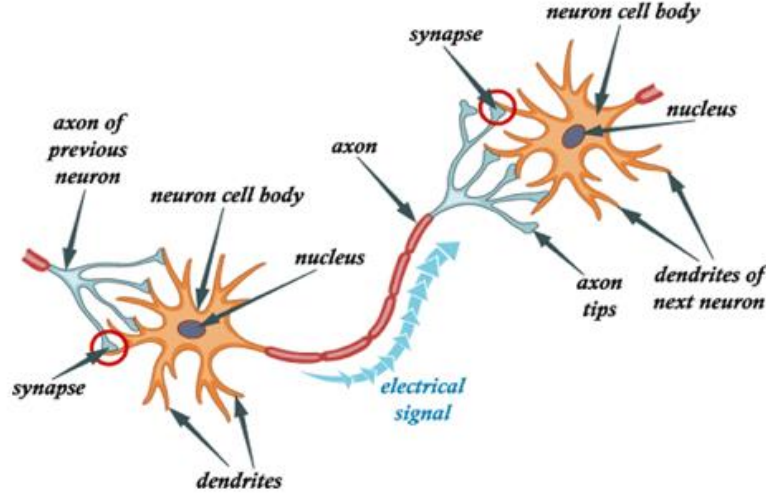


Figure 2.2: Schematic diagram of biological neurons [14]

A biological neural network is a network of neurons connected by synapses to perform a certain task. Multiple neural networks are interconnected to form large scale brain networks. A brain may contain about 86 billion neurons and more than 100 trillion synaptic connections, [14].

2.2.2 An Artificial Neuron Model

An artificial neuron is a mathematical function meant to simulate the behavior of a biological neuron. It is also known as perceptron. It consists of summation, multiplication and activation function, as seen in Figure 2.3. The output of artificial neuron is given by:

$$y_j = \varphi \left(\sum_{k=0}^n w_{kj} x_k \right) \quad (2.1)$$

where, y is the output, x is the input and φ is the activation function. j and k are the indices for the neural network layer and the artificial neuron in the layer.

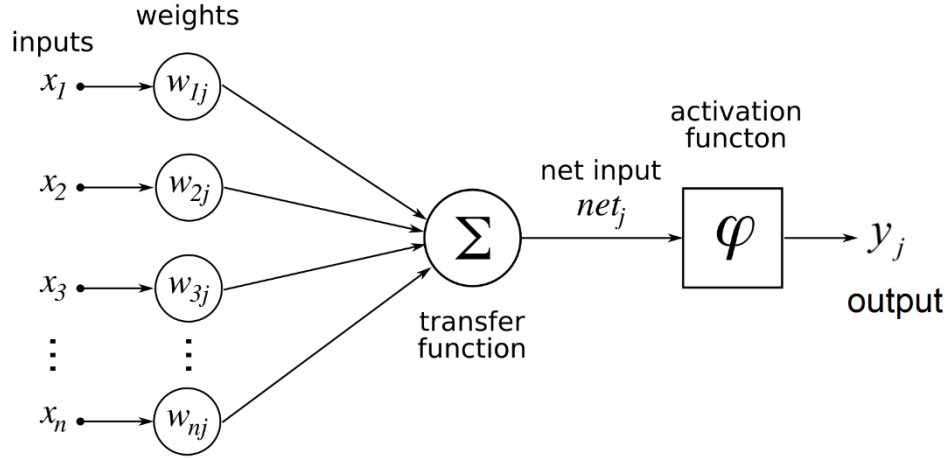


Figure 2.3: A simple artificial neuron model [15]

The output of an artificial neuron is equivalent to the output of a biological neuron. The output value may be transmitted to the next layer of neurons but may also be transmitted as an output of the neural network as a part of the output vector. By itself, it has almost no learning capabilities, [12]. The weights are calculated, and the activation function is predetermined as per the application.

Multiple neurons can be combined to form a single layer of neural network. Multiple layers are stacked next to each other to form a complete Artificial Neural Network (ANN), as seen in Figure 2.4. An ANN can be trained to perform complex tasks.

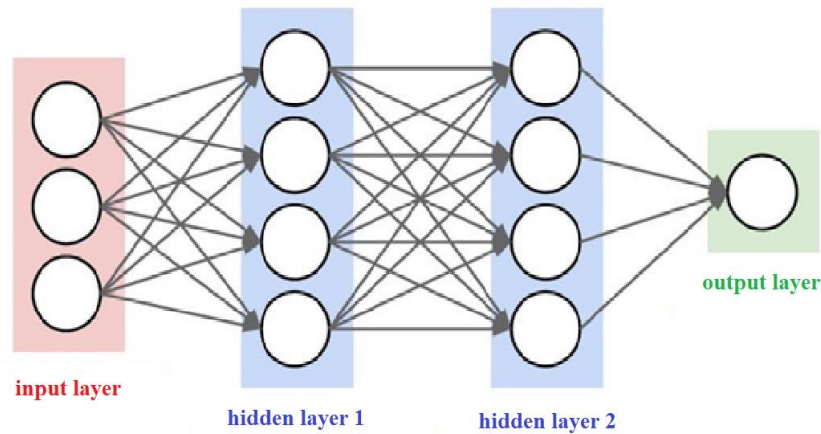


Figure 2.4: A simple Artificial Neural Network [12]

Using enough data, an ANN can learn patterns in numerical data and train its weights to emulate the system that represent the given data. With this, it can model complex patterns and predict output of high-level systems. It can be used with anomaly detection, natural language processing, image classification, and forecasting, [14].

2.2.3 Back-propagation

Back-propagation is the shorthand for the backward propagation of errors. It is used in artificial neural networks to determine the gradient needed to calculate the weights of the network. It is a popular neural network learning algorithm because it is conceptually simple, computationally efficient and often, works.

Back propagation works in two stages:

1. Feedforward Propagation

In feedforward propagation, the training data is fed through the network and the output is calculated. In this phase, initially the weights of the network are randomly generated, and they are not changed.

2. Feedback Propagation

In feedback propagation, output from the randomized network is compared with the desired output. The error is calculated by subtracting calculated output from desired output. The error is then backpropagated through the multiple layers of the neural network while minimizing errors on every layer. This results in updating weights of all the layers in the neural network.

These two stages are repeated multiple times until the error is within satisfactory limits.

2.2.4 Theoretical background of back-propagation

While the following derivation primarily focuses on application for artificial neural networks, it can also be applied for other gradient-based learning methods, [16].

Assume a multilayer feedforward artificial neural network M . The cost function of the neural network is given by, [16]:

$$E^p = C(D^p, M(Z^p, W)) \quad (2.2)$$

where, Z_p is the input vector, W is the weight vector, D^p is the desired output and p is the input index.

A neural network is simply a stack of modules each of which implements the following function, [16]:

$$X_n = F(W_n, X_{n-1}) \quad (2.3)$$

where, X_n is the vector representing output of the module, W_n is the vector representing the weights of the module (a subset of W), X_{n-1} is the module representing the input vector as well as the output vector of previous module and input X_0 to the first module is the input pattern Z_p .

If the partial derivative of E^p with respect to X_n can be calculated, the partial derivatives of E^p with respect to W_n and X_{n-1} can be calculated using backward recurrence:

$$\frac{\delta E^p}{\delta W_n} = \frac{\delta F}{\delta W}(W_n, X_{n-1}) \frac{\delta E^p}{\delta X_n} \quad (2.4)$$

$$\frac{\delta E^p}{\delta X_{n-1}} = \frac{\delta F}{\delta X}(W_n, X_{n-1}) \frac{\delta E^p}{\delta X_n} \quad (2.5)$$

where, $\frac{\delta F}{\delta W}(W_n, X_{n-1})$ is the Jacobian of F with respect to W at point (W_n, X_{n-1}) and

$\frac{\delta F}{\delta X}(W_n, X_{n-1})$ is the Jacobian of F with respect to X . When these equations are applied to all the

layers in reverse order, the partial derivatives of the cost functions with respect to all the parameters can be computed. This is called back-propagation, [16].

In traditional neural network, each layer also has an activation function, for example: a sigmoid function.

$$Y_n = W_n X_{n-1} \quad (2.6)$$

$$X_n = F(Y_n) \quad (2.7)$$

where, F is the activation function, W_n is the weight vector, X_{n-1} is the input vector of the layer, X_n is the output vector and Y_n is the vector of weighted sums. Using the chain rule on above equations, one can obtain the following, [16]:

$$\frac{\partial E^p}{\partial y_n^i} = f'(y_n^i) \frac{\partial E^p}{\partial x_n^i} \quad (2.8)$$

$$\frac{\partial E^p}{\partial w_n^{ij}} = x_{n-1}^j \frac{\partial E^p}{\partial y_n^i} \quad (2.9)$$

$$\frac{\partial E^p}{\partial x_{n-1}^k} = \sum_i w_n^{ik} \frac{\partial E^p}{\partial y_n^i} \quad (2.10)$$

The above equations can be written in matrix form as:

$$\frac{\partial E^p}{\partial Y_n} = F'(Y_n) \frac{\partial E^p}{\partial X_n} \quad (2.11)$$

$$\frac{\partial E^p}{\partial W_n} = X_{n-1} \frac{\partial E^p}{\partial Y_n} \quad (2.12)$$

$$\frac{\partial E^p}{\partial X_{n-1}} = W_n^T \frac{\partial E^p}{\partial Y_n} \quad (2.13)$$

A simple gradient descent algorithm for weight adjustment can be derived as, [16]:

$$W(t) = W(t-1) - \eta \frac{\partial E}{\partial W} \quad (2.14)$$

It can also be written as:

$$W(t+1) = W(t) - \eta \frac{\partial E}{\partial W} \quad (2.15)$$

where, η is the learning or adaptation rate. In its simplest form, it is a scalar constant. However, it may also be a diagonal matrix.

2.2.5 Activation functions

Activation function can be defined as the transfer function of a node in a neural network. It may introduce non-linearity into the model which can make the neural network more flexible in modeling and classification. The types of activation functions can be divided into two groups:

a. Linear Activation Function

A linear activation function is also known as the identity function. It can be visualized as a line. Using this activation function, output of a node isn't restricted in any range. While this activation function is simplest to implement, it doesn't help with the complexity and non-linearity of usual kinds of data normally fed through a neural network, [17].

Each layer of the neural network is activated by a linear function. The linearity is then transferred to the next layer which in turn gets transferred on. Therefore, with linear activation, the neural network ends up being a simple linear function as shown in Figure 2.5. This means, the neural network can be replaced by a single linear layer. This removes all the advantage of stacking multiple layers for deep learning, [17].

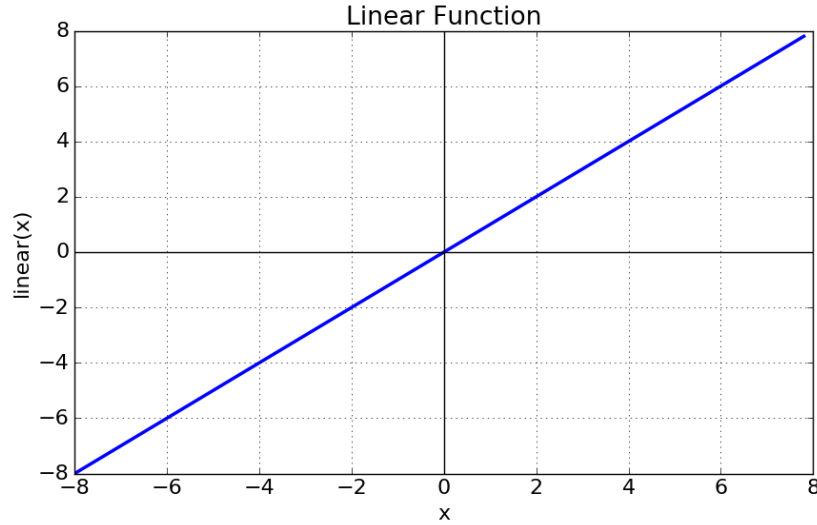


Figure 2.5: Linear activation function [18]

The equation is given by: $f(x) = x$

The range of output is from -infinity to infinity.

b. Non-linear Activation Function

Non-linear activation functions are the most used functions in machine learning. Non-linearity helps the neural network adapt and generalize to different types of data, [17]. Different types of activation functions are used for different applications. Some of them are listed below:

1. Sigmoid Activation Function

A sigmoid function is also known as a logistic activation function. The mathematical equation is given as:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.16)$$

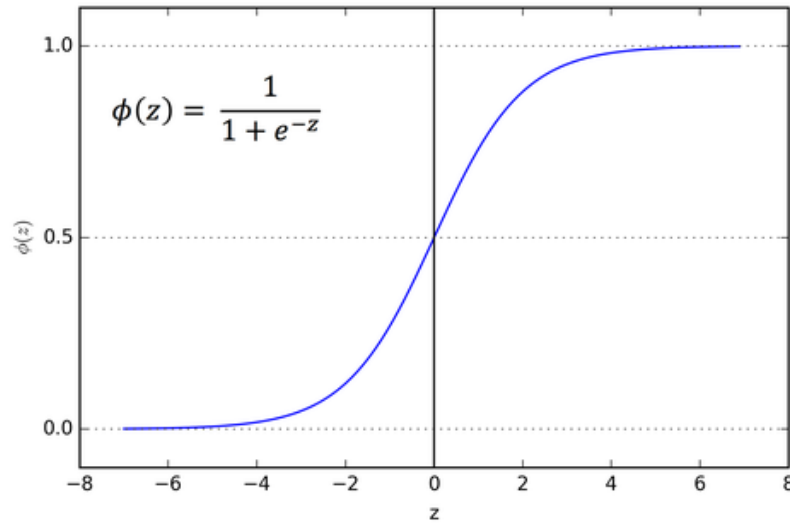


Figure 2.6: Sigmoid activation function [18]

The main advantage of the sigmoid function is its range. The value of the function is bounded by $[0,1]$ as seen in Figure 2.6. Therefore, it is especially used in models where a probability is being predicted. Since probability ranges from 0-1, sigmoid makes a good fit. For a multiclass problem i.e., classification for multiple classes, a generalized logistic function called SoftMax function is used, [17].

The sigmoid function is monotonic and differentiable. However, its derivative is not monotonic. This can cause a neural network to get stuck during training. At the higher values of x , the function tends to not respond to changes in x . This can cause a vanishing gradient problem. The network may learn very slowly or stop learning altogether. However, there are workarounds to the problem and the sigmoid function is still widely used in classification problems, [17].

2. Tanh or Hyperbolic tangent activation function

Tanh or Hyperbolic tangent activation function is very similar to the sigmoid function but has a range of $[-1,1]$. It also has a higher gradient than the sigmoid function as shown in Figure 2.7. The negative inputs are mapped strongly negative and near zero inputs are mapped close to zero.

The mathematical equation is given by:

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2\text{sigmoid}(2x) - 1 \quad (2.17)$$

Hyperbolic tangent activation functions are generally used to classify between two classes as it can strongly distinguish between negative and positive inputs. However, it can fall victim to similar issues as sigmoid function like the vanishing gradient problem, [17].

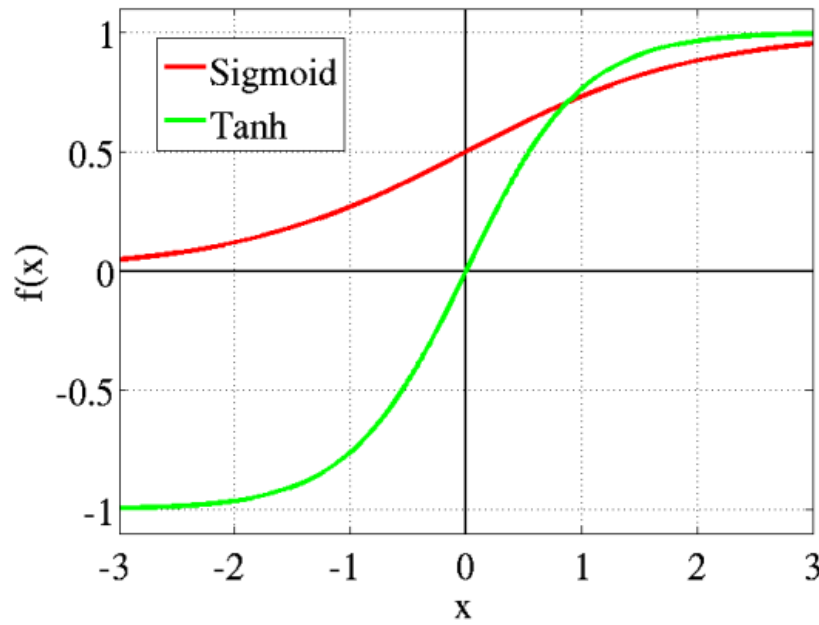


Figure 2.7: Hyperbolic Tangent activation function [18]

3. Softmax activation function

Softmax activation function is generally used in multi-class classification problems. One of the advantages of softmax is that it arranges the output such that it provides the probability for each class, [19]. The output ranges from (0-1) and the sum of all outputs is always 1. As such, it is always in the output layer of a neural network.

It is defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}} \quad (2.18)$$

where, x_i and x_j are the i^{th} and j^{th} output of the output layer with n outputs.

4. Rectified Linear Unit (ReLU) activation function

Rectified Linear Unit activation function is currently the most used activation function. ReLU is computationally less expensive than tanh or sigmoid functions. It is used in almost all CNNs or deep learning, [17]. It is defined as:

$$f(x) = \max(0, x) \quad (2.19)$$

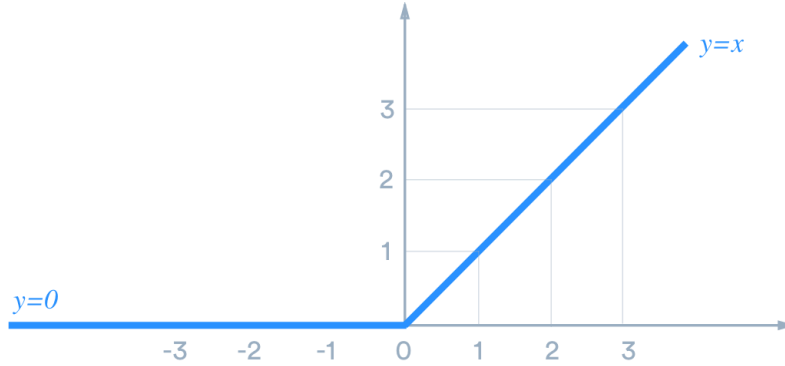


Figure 2.8: Rectified Linear Unit activation function [18]

While similar to linear activation, it is non-linear in nature. As seen in Figure 2.8, it blocks all negative inputs and passes positive values as is, [18]. It can act as a good approximator for any function. However, it is not a bound function, $[0, \infty)$. This means the node outputs may blow up after activation.

Since it filters negative inputs, it causes a sparsity in activations, i.e. not all neurons activate in the network. However, this can bring about the dying ReLU problem. Since it filters all negative values, its gradient can be zero for some datasets. This means, it may not adapt to some dataset

which has predominantly negative values. This can be solved using the leaky ReLU activation function, [18]. In leaky ReLU, instead of completely filtering the negative values, it has a slightly horizontal line instead of a horizontal line for $x < 0$. This is depicted in Figure 2.9.

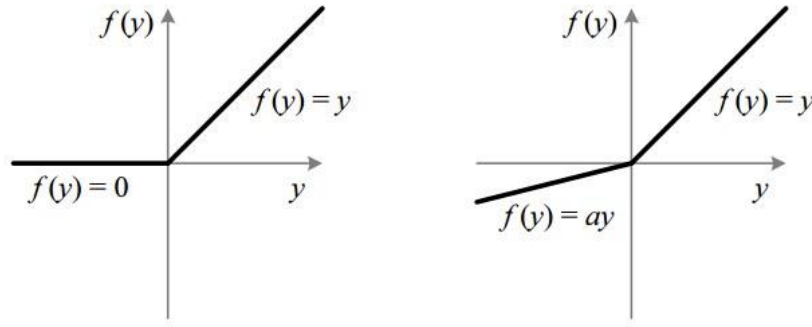


Figure 2.9: ReLU and Leaky ReLU [18]

2.2.6 Cost Functions

A cost function is a measure of how accurate or good a neural network is with respect to its training and test samples. It compares output of the neural network with the expected output. It may depend on various parameters such as weights and biases. A cost function is a single value, even for multi-class neural networks, as it is a measure of how good a neural network is instead of how good each output is.

A cost function should have following properties:

1. The average of the cost function must be computable.

$$C = \frac{1}{n} \sum_x C_x \quad (2.20)$$

where, C_x is the cost of each output and n is the number of outputs.

This allows for computation of the gradient with respect to weights and biases for a single example and application of the gradient descent method.

2. The cost function should be independent of any activation values of a neural network except its output. This restriction is applied for the sake of backpropagation. If the cost function was dependent on anything other than the output values, the backpropagation wouldn't be applicable. With this, only the gradient of the last layer is dependent on the cost function while the other layers dependent on the next layer.

2.2.6.1 Types of cost functions

1. Quadratic cost

Quadratic cost is also known as mean squared error, maximum likelihood or sum of squared errors. It calculates the average of square of errors. It is always positive. The values closer to zero are considered better.

It is given by:

$$C_{MSE} = \frac{0.5}{n} \sum_{i=1}^n (Y_i - \bar{Y}_i)^2 \quad (2.21)$$

where, Y_i is the desired output and \bar{Y}_i is the calculated output.

The gradient of the cost function with respect to output of the neural network is given by:

$$\nabla C_{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \bar{Y}_i) \quad (2.22)$$

2. Cross entropy cost

Cross entropy is one of the most widely used cost functions in ANNs. It is defined as, [20]:

$$C_{cross_entropy} = -\frac{1}{n} \sum_{i=1}^n Y \log(\bar{Y}) - (1 - Y) \log(1 - \bar{Y}) \quad (2.23)$$

where, Y is the desired output and \bar{Y} is the neural network output.

A special class of cross-entropy, binary cross entropy is used for classification, [20]. In binary cross entropy:

$$C_{binarycrossentropy} = \begin{cases} -\log(\bar{Y}), & \text{if } Y = 1 \\ -\log(1 - \bar{Y}), & \text{if } Y = 0 \end{cases} \quad (2.24)$$

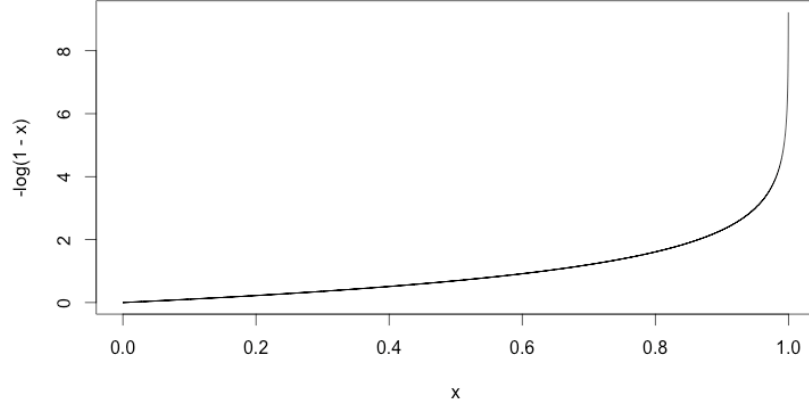


Figure 2.10: Binary cross entropy function, Y=0 [20]

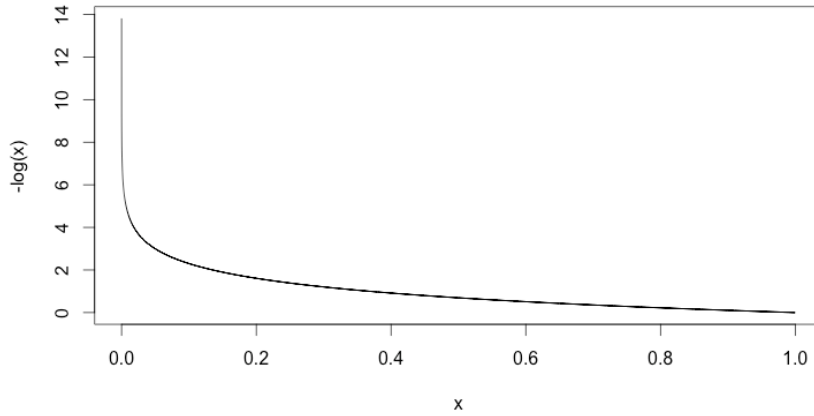


Figure 2.11: Binary cross entropy function, Y=1 [20]

From Figure 2.10 and Figure 2.11, it can be seen that the cross-entropy converges towards zero when the expected value is close to the output value. It also spikes in value the farther the output gets from the expected value. This is a highly desirable property of a cost function. It can also help alleviate the vanishing gradient problem of sigmoid and tanh activation functions, [20]. If the output layer of a neural network is activated using the sigmoid function, it is almost always advisable to use cross entropy as the cost function.

Other than the ones discussed above, there are other cost functions such as exponential cost, Hellinger Distance, Kullback-Leibler Distance and Itakura-Saito Distance.

2.2.7 Weight initialization

The output of a perceptron is given by:

$$y_j = \varphi \left(\sum_{k=0}^n w_{kj} x_k \right) \quad (2.25)$$

where, y is the output, x is the input and φ is the activation function. j and k are the indices for the neural network layer and the artificial neuron in the layer.

Weights of the perceptron play a huge part in the learning capability of a perceptron and therefore, the entire neural network. A neural network “learns” by adapting its weights to the input data. Therefore, initial values of weights may determine speed and accuracy of the neural network.

There are multiple ways to initialize the weights of a neural network:

1. Zero initialization

Initialize all weights to zero. This causes the output of each neuron and subsequently output of the neural network to be zero. Since the gradient of loss function is the same for all layers, the weights are adapted by the same value in next iteration. This continues on for all iterations and the consequent neural network has symmetric hidden layers. This makes it no better than linear approximation of given data, [21].

2. Random initialization

Random weight initialization is better than zero weight initialization. This assures that the layers aren't identical in nature and provides non-linearity for better adaption of weights. However, issues can arise when all the weights are initialized very high or very low. For low value initializations, similar issues as zero initialization may occur. For high value initialization, the

neural network may suffer from the vanishing gradient problem. Many solutions have been proposed for this such as He initialization and Xavier initialization, [21].

2.3 Convolutional Neural Network

Convolutional Neural Network (CNN) is one of the most widely used neural network architecture today. It is generally used for image classification, image segmentation and object detection problems. A CNN normally takes in an order three tensor as input, image with height H , width W and three channels (R, G, B color channels). However, higher order tensors can also be used, [22]. The input is then passed through a series of layers for processing. There are multiple types of layers of which convolution, ReLU activation and pooling layers are an integral part of most CNNs.

The main advantage of Convolution Neural Networks over other image processing techniques is that it requires very little image pre-processing as it learns all the necessary filters during training. This independence from hand-tailored filters is a major selling point for CNNs.

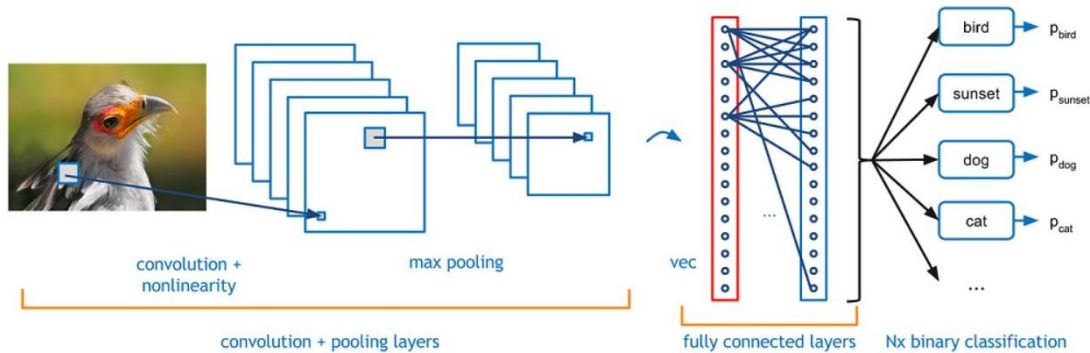


Figure 2.12: Structure of Convolutional Neural Network [23]

2.3.1 Convolution Layer

The convolution layer is always the first part of a CNN. It is also the most important layer in a CNN, thus the name convolutional neural network.

2.3.1.1 Convolution

A convolution kernel is overlaid on top of the input matrix (or image) and the product between the numbers of the same location in kernel and input is calculated. A singular number is then obtained by summing all the products together. The kernel is then moved down by one element. These steps are repeated until it reaches the bottom border of the input matrix. The kernel is again moved to the top and moved to the right by one element. All these steps are repeated again until the kernel has moved to the bottom right of the matrix, [22].

For example:

$$\begin{pmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 16 & 11 \\ 24 & 28 & 17 \end{pmatrix}$$

Step 1: $1 \times 1 + 1 \times 4 + 1 \times 2 + 1 \times 5 = 12$

Step 2: $1 \times 4 + 1 \times 7 + 1 \times 5 + 1 \times 8 = 24$

For order three tensors, the convolution process is done similarly. Suppose, an input order three tensor of size $H^l \times W^l \times 3$. Then, the convolution kernel overlaid on the input tensor will be of size $H \times W \times 3$. The convolution is performed with the same steps as the convolution between two simple matrices as described above. The kernel is moved from top to bottom and left to right to complete the convolution.

In the convolution layer, multiple convolution kernels are used. Assuming D kernels are used, the final output of the convolution layer is of size $(H^l - H + 1) \times (W^l - W + 1) \times D$. All the kernels are grouped into an order four tensor f of size $H \times W \times D^l \times D$.

Another important concept in convolution is called stride. In the above example, the kernel is overlaid with the input matrix at all possible spatial locations, which corresponds to the stride

s=1. . However, if stride is increased (s>1), the kernel will skip s-1 pixel locations. While this creates a smaller output, details from the input may be lost, [22].

Assuming a stride of 1, the convolution layer can be expressed in the following equation:

$$y_{i^{l+1}, j^{l+1}, d} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \sum_{d^l=0}^{D^l-1} f_{i,j,d^l,d} \times x_{i^{l+1}+i, j^{l+1}+j, d^l}^l \quad (2.26)$$

The above equation is repeated for all $0 \leq d \leq D^{l+1}$, and for any spatial location (i^{l+1}, j^{l+1}) that satisfies the following conditions:

$$\begin{aligned} 0 \leq i^{l+1} &< H^l - H + 1 = H^{l+1} \\ 0 \leq j^{l+1} &< W^l - W + 1 = W^{l+1} \end{aligned}$$

In this equation, $x_{i^{l+1}+i, j^{l+1}+j, d^l}^l$ is the element of x^l indexed by $(i^{l+1}+i, j^{l+1}+j, d^l)$.

Higher values mean that the patterns in the kernel matches that part of the image more closely than the ones with lower values. This is called feature matching and the kernels are also called feature detectors. As the name suggests, each feature detector represents a separate feature. The output of the feature detectors is also called feature maps, [23].

2.3.1.2 Significance of convolution

The convolution kernels can highlight different features of the image. For example, let us take the following convolution kernel:

$$K = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

When this kernel is convolved over Figure 2.13(a), it highlights horizontal edges of the image as shown in Figure 2.13(b). If the kernel is transposed and then convolved over the image, it highlights the vertical edges as shown in Figure 2.13(c).

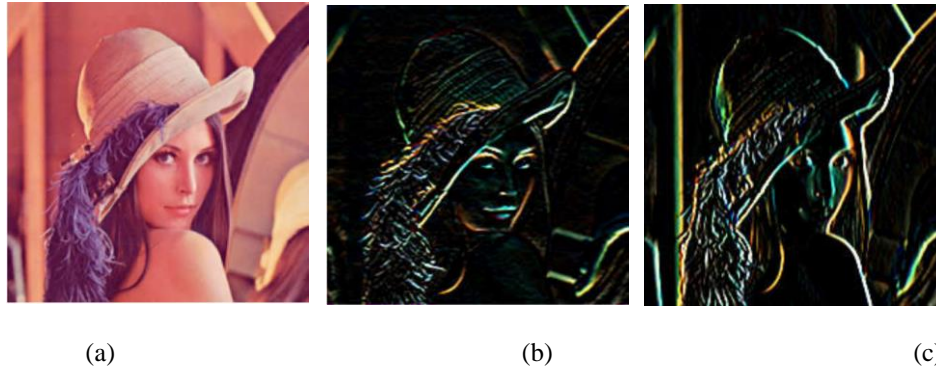


Figure 2.13: (a) Lenna Image, (b) Horizontal Edges Highlight, (c) Vertical Edges Highlight [22]

As the CNN gets deeper, it can learn more complex patterns such as curves, triangles or even cat's head. And since each convolution kernel is overlaid over the whole image, it doesn't matter where the pattern exists in the input image. For example, if multiple cats appear in an image, the same "cat head like pattern" may be activated at multiple locations. It also means that no matter how the cat is positioned, a CNN will detect its head pattern, [22].

In a deep neural network, convolution also helps parameter sharing among different layers. For example, if a CNN has learned to identify "dog head like pattern" and "cat head like pattern", it doesn't need to devote two different kernels for them. The CNN may learn "eye-like pattern" and "animal-fur-texture-pattern" which are shared by both dog and cat head patterns. Combinations of kernels like these can form deep and hierarchical structures very effective at learning features from images for visual recognition, [22].

A key concept of CNN, and also deep learning, is distributed representation. In a classifier of N objects, a CNN may extract M features from a single object. However, all M features may be useful for classifying all N objects, [22].

2.3.2 ReLU layer

ReLU activation is applied to the output of the convolution layer. The purpose of this layer is to increase the non-linearity of the CNN. Since the input of CNN, i.e. images, are non-linear in nature, non-linearity must be maintained in the output of the convolution layer, [22].

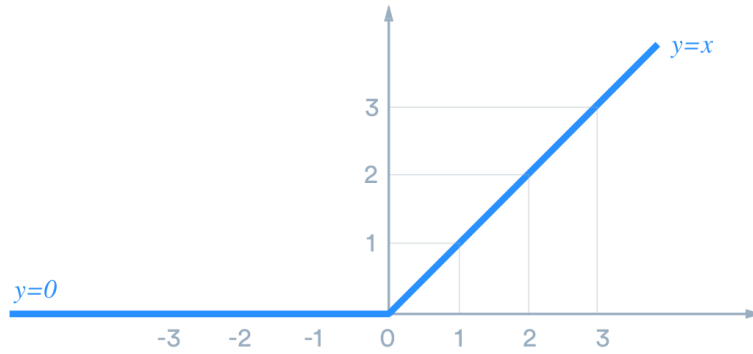


Figure 2.14: ReLU function [18]

This property is useful for recognizing complex patterns and objects. For example, instead of using the shape of the head of a cat as the evidence for “presence of cat” in an image, it may also detect other features such as torso, fur, legs, etc. This can provide higher confidence for “presence of cat” in the image, [22].

2.3.3 Pooling

The pooling layer pools multiple values in the feature maps generated by the convolution and ReLU layers into a single value. This reduces the number of data points, thus reducing the size of neural network while not losing anything of significant value.

Let x^l be the input, order three tensor of size $H^l \times W^l \times D^l$, of the pooling layer (l^{th} layer). Pooling layer requires no weights so there is no training done for this layer. The size of pooling operation ($H \times W$) is specified in the CNN structure. Let us suppose H^l is divisible by H and W^l is divisible by W and the stride of the pooling operation is equal to the pooling spatial extent. That is, the strides in vertical and horizontal direction are H and W respectively. The most

commonly used pooling setup is $H=W=2$ with a stride of 2. Then, the size of the output tensor y or x^{l+1} , H^{l+1} , W^{l+1} and D^{l+1} can be calculated as, [22]:

$$H^{l+1} = \frac{H^l}{H}, W^{l+1} = \frac{W^l}{W}, D^{l+1} = D^l$$

There are two main types of pooling:

a. Max Pooling

In max pooling, the pooling operation maps a subregion into its maximum value. Since higher value means higher similarity with a pattern in a feature detector, it is basically acknowledging the presence of that pattern in the image while discarding the unnecessary values which signifies the lack of similarity with the pattern, [22].

It is mathematically defined as:

$$y_{i^{l+1}, j^{l+1}, d} = \max_{0 \leq i < H, 0 \leq j < W} x_{i^{l+1} \times H + i, j^{l+1} \times W + j, d}^l \quad (2.27)$$

where, $0 \leq i^{l+1} < H^{l+1}$, $0 \leq j^{l+1} < W^{l+1}$, and $0 \leq d < D^{l+1} = D^l$

Example:

$$\begin{pmatrix} 0 & 4 & 1 & 7 \\ 3 & 2 & 5 & 1 \\ 1 & 3 & 0 & 0 \\ 1 & 2 & 2 & 1 \end{pmatrix} \xrightarrow{2 \times 2 \text{ max pooling}} \begin{pmatrix} 4 & 7 \\ 3 & 2 \end{pmatrix}$$

b. Average Pooling

In average pooling, the pooling operation maps a subregion into its average value, [22]. The output of average pooling represents the average similarity between the image and a feature detector.

$$y_{i^{l+1}, j^{l+1}, d} = \frac{1}{HW} \sum_{0 \leq i < H, 0 \leq j < W} x_{i^{l+1} \times H + i, j^{l+1} \times W + j, d}^l \quad (2.28)$$

where, $0 \leq i^{l+1} < H^{l+1}, 0 \leq j^{l+1} < W^{l+1}$, and $0 \leq d < D^{l+1} = D^l$

Example:

$$\begin{pmatrix} 0 & 4 & 1 & 7 \\ 3 & 2 & 5 & 1 \\ 1 & 3 & 0 & 0 \\ 1 & 2 & 2 & 1 \end{pmatrix} \xrightarrow{2 \times 2 \text{ average pooling}} \begin{pmatrix} 2.25 & 3.5 \\ 1.75 & 0.75 \end{pmatrix}$$

2.3.4 Flattening

The flattening layer converts the output from convolution and pooling layers into a one column matrix. So, it basically takes the feature maps generated from the previous layer and restructures them into a $n \times 1$ matrix where n is the number of data points generated.

Example:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

2.3.5 Full Connection

A full connection layer is an ANN attached at the end of the CNN as shown in Figure 2.12. This layer takes in the input from its preceding layers (convolution, ReLU and pooling) and outputs N values (an N dimensional vector) where N is the number of classes the network can choose from. The output may be represented in different ways depending on the activation function and cost function used in the full connection layer, [22]. Figure 2.15 shows the structure of a simple fully connected layer.

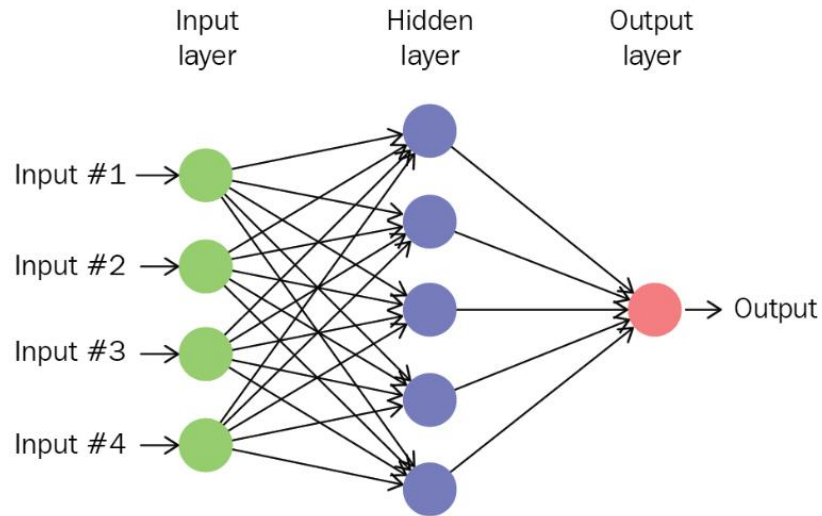


Figure 2.15: Fully Connected Layer [33]

Assuming the output of the flattening layer is same as the example in previous section, the inputs to the fully connected layer, Input 1, 2, 3 and 4 would be 1, 0, 0 and 1 respectively.

2.4 Keras

Keras is a high-level neural network API in Python. It can use popular machine learning and mathematical libraries such as TensorFlow, Theano or CNTK. It was designed to speed up the process of implementation and experimentation of neural networks, [24].

The advantages of Keras are, [24]:

1. Easy and fast prototyping due to its accessibility, modularity and extensibility.
2. Runs on both CPU and GPU.
3. Supports different types of neural networks such as CNN and Recurrent Neural Network (RNN) or the combination of both.

2.4.1 Keras Installation

Keras can be installed in many ways depending on the python editor that is used. It also has a few dependencies that need to be installed before it can be used. The dependencies are TensorFlow, Theano and CNTK, depending on the preferred backend. There are other optional

dependencies: cuDNN, HDF5, h5py, graphviz and pydot. Keras can be installed for most virtual environments by using pip command:

pip install keras

For Anaconda, Keras can be installed using conda command:

conda install keras

2.4.2 Basic functions of Keras

The core data structure of Keras is a model which defines how layers are organized. Sequential model is used for this research. Sequential and some core functions are described below.

- a. Sequential: It builds a linear stack of layers.
- b. `.add()`: It stacks layers to the model. For example: convolution, fully connected, pooling, etc.
- c. `.compile()`: It configures the learning process of the model with hyperparameters such as optimizer, loss function and metrics of evaluation.
- d. `.fit()`: Its fits the model to the training set and validates the model with the given test set.

The number of epochs and batch size is also defined with this function.

- e. `.predict()`: It predicts the output of the model using the given input.

Example of the usage of these functions are detailed in Chapter 3.

2.5 Kinect

Kinect, coined from the words ‘kinetic’ and ‘connect’, is a sensor device by Microsoft™. When it was released in 2010, Microsoft™ planned to market it primarily for gaming. However, numerous developers, tech enthusiasts and researchers soon developed new applications to use Kinect with.

The Kinect used in this project is the original Kinect for XBOX 360. Recently, Microsoft™

has released another from the Kinect line/family, Project Kinect for Azure, which it plans to deploy for machine learning applications. Azure is Microsoft's branding for all things AI. What this thesis project is attempting to do can be likened to a small portion of what Project Kinect for Azure is attempting to achieve.

Kinect 360 was used to capture images in this project. It is essentially a little black rectangular box with an RGB camera, a depth sensor and an array of four microphones. The RGB camera detects red, green and blue color components, hence the name RGB. The depth sensor is a combination of a monochrome CMOS sensor and ultraviolet radiation that projects a 3D imagery of the environment in front of the Kinect.

2.6 Kinect control using Python

Kinect is controlled using Python using Open Natural Interaction 2 (OpenNI2) and Open Computer Vision (OpenCV) libraries.

OpenNI2 is an open source software project designed to operate and interact with Natural Interaction (NI) devices, [25]. Natural Interaction devices are devices that can capture body movements and sounds that allows the user to interact with computers more naturally. Some of these devices are Kinect and Wavi X-tion. OpenNI2 is written in C with Python wrappers being available. OpenNI2 accesses the RGB or depth sensor in the Kinect and sends the video stream to the computer.

Open Source Computer Vision Library is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products, [26]. OpenCV captures frames from the video stream accessed using OpenNI2.

The code for RGB image capture is included in the Appendix.

2.7 Wheelchair

The wheelchair used for this project is PermobilTM M300 Corpus HD. It is a heavy duty powered wheelchair designed to carry up to 450lbs. It can reach a maximum speed of 5.1 mph. The wheelchair is fitted with two batteries sized 74Ah, each with a nominal voltage of 12 Volts. The electronics are safeguarded by a circuit breaker of 63A. The power usage of the wheelchair is monitored by ExtechTM PM120 Power Monitor. The wheelchair is controlled using a joystick control system attached to the right arm rest. The joystick module is R-net CJSM-sw manufactured by Curtiss-WrightTM, [31].

3. EXPERIMENTAL SETUP AND IMPLEMENTATION OF NEURAL NETWORK

3.1 Experimental Setup

3.1.1 Block Diagram

Figure 3.1 shows the block diagram of the setup used for image capture and transfer from Kinect to the computer. The computer is also connected to an Arduino which controls the joystick control mechanism for the wheelchair joystick.

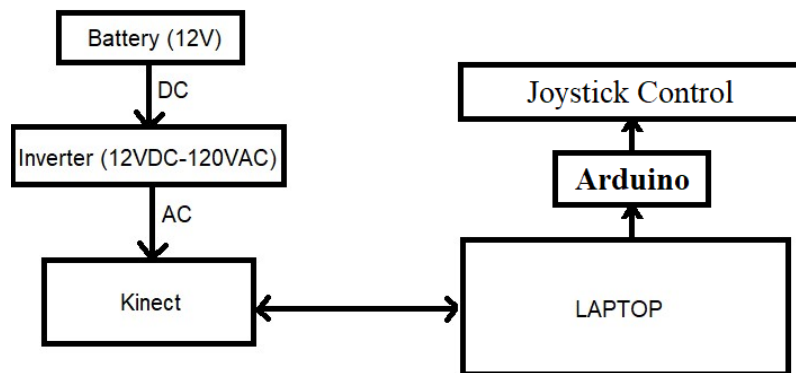


Figure 3.1: Block diagram of the experimental setup

3.1.2 Current Setup for Image Capture

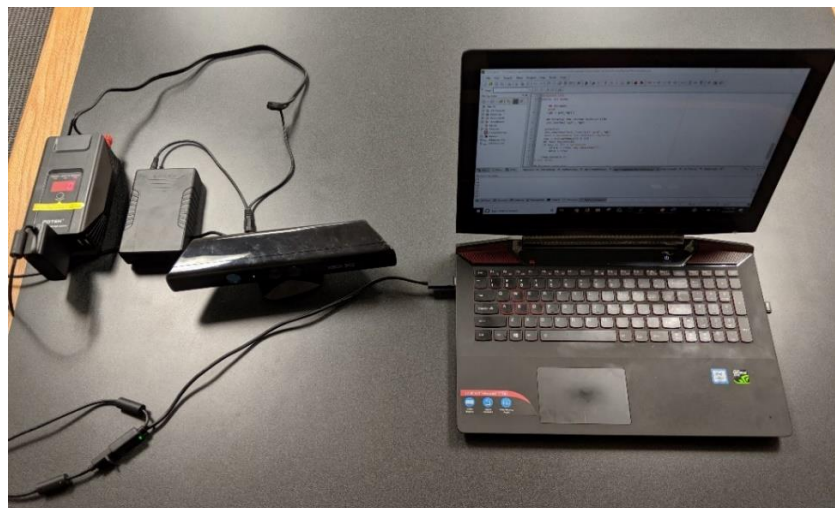


Figure 3.2: Setup for Image Capture

Figure 3.2 shows the experimental setup for the project. The Kinect is powered by a 120V 60Hz inverter which in turn gets its power from a 12V battery. The Kinect is then connected to a laptop which classifies the images captured by the Kinect.

3.1.3 Computer Specification

Processor: Intel® Core™ i7-6700HQ CPU @ 2.60Hz (8 CPUs), ~2.GHz

RAM: 8192 MB

Operating System: Windows 10 Home 64-bit (10.0, Build 17134)

GPU: NVIDIA® GeForce® GTX 960M

Average training time for CNN for one epoch: 7 minutes

3.2 Artificial Neural Network

While a CNN is designed for image classification, it is still built on top of an ANN(fully connected layer). Therefore, to understand and demonstrate the working of an ANN, a simple prediction model was trained using a churn dataset found in Kaggle, [29].

3.2.1 Predictive Churn Model

A churn is defined as the ratio of customer cancellation to the number of active customers. A predictive churn model or churn modeling is used to measure the immediate and future risk of customer cancellation. The probability of churn can be predicted using statistical and machine learning techniques.

3.2.2 Dataset and Data preprocessing

RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
1	15634602	Hargrave	619	France	Female	42	2	0	1	1	1	101348.88	1
2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	1	112542.58	0
3	15619304	Onio	502	France	Female	42	8	159660.8	3	1	0	113931.57	1
4	15701354	Boni	699	France	Female	39	1	0	2	0	0	93826.63	0
5	15737888	Mitchell	850	Spain	Female	43	2	125510.8	1	1	1	79084.1	0

Figure 3.3: Sample of dataset for churn modeling

Figure 3.3 shows a sample of the dataset used for churn modeling. The data represents

customers of a fictional bank in Europe. The data fields are “RowNumber”, “CustomerID”, “Surname”, “CreditScore”, etc. The data field “Exited” provides information regarding whether the customer has left the bank, “1” for customer who have left and “0” for those who haven’t.

Before the data is used for training a neural network, it is preprocessed to remove redundant data and encode labels such as “Geography” and “Gender” into usable values.

“RowNumber”, “CustomerID” and “Surname” have no influence over churn and are therefore discarded. Labels such as “Geography” and “Gender” are encoded into binary values for usage in neural network as they may influence the customer from leaving or staying with the bank. For this neural network, they are encoded as:

Geography: France = 0 0

Spain = 0 1

Germany = 1 0

Gender: Male = 1

Female = 0

Since “Geography” has now been encoded into two bits, the total number of input variables increases to eleven.

Finally, parameters like “CreditScore”, “Age”, “Balance” and “EstimatedSalary” are standardized into similar range as the rest of the parameters. In their original form, these parameters may dominate other parameters in the neural network as they are much larger in comparison. Therefore, the values of these parameters are scaled in the range of 0 to 1.

The dataset is also divided into two parts, training set and test set. The training set (X_{train}, Y_{train}) is used to train the weights of the model and the test set (X_{test}, Y_{test}) is used to determine the accuracy of the trained model. This ensures that the trained model isn’t only

accurate for the training data and can fit data outside of the set used to train it. Normally the data is divided by a ratio 4:1 or 9:1 depending on the size of the dataset.

3.2.3 ANN implementation using Keras

The ANN code was written in Anaconda using Spyder IDE in Python 3.6.7. It was executed in both Spyder and Pyscripter IDE.

```
#Importing the Keras libraries and packages
import keras
from keras.models import Sequential
from keras.layers import Dense

#Initialising the ANN
classifier=Sequential()

#Adding the input layer and the first hidden layer
classifier.add(Dense(output_dim=6, init='uniform',activation='relu', input_dim=11))

#Adding the second hidden layer
classifier.add(Dense(output_dim=6, init='uniform',activation='relu'))

#Adding the output layer
classifier.add(Dense(output_dim=1, init='uniform',activation='sigmoid'))

#Compiling the ANN
classifier.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])

#Fitting the ANN to the Training Set
classifier.fit(X_train, Y_train, batch_size=10,nb_epoch=10)

#Predicting the Test set results
Y_pred=classifier.predict(X_test)
Y_pred= (Y_pred>0.5)

#Making the Confusion matrix
from sklearn.metrics import confusion_matrix
cm=confusion_matrix(y_test,y_pred)
```

Figure 3.4: Artificial Neural Network Code in Keras

Figure 3.4 shows the code for the implementation of ANN in Python using Keras. Each line is discussed below:

1. **classifier = Sequential()**

This initiates a sequence of layers named classifier. Initially, the defined sequence is empty. The layers that are defined after this function call are added sequentially to the neural network.

2. **classifier.add()** adds a layer to the neural network. “Dense” layer means each of the neuron of the layer is connected to all the inputs for that layer. This is also called a fully connected layer.

a. Input layer: **classifier.add(Dense(output_dim=6, init='uniform', activation='relu', input_dim=11))**

Since there are 11 independent variables chosen to model the churn process, the number of inputs for the input layer is taken as 11 (input_dim=11). The input layer has 6 neurons and therefore, it has 6 outputs (output_dim=6). The activation function is chosen as ReLU which is considered one of the best activation functions for input and hidden layers (activation='relu'). The weights are initialized in uniform random distribution (init='uniform'). This also ensures that all the weights are initialized to a small value.

b. Hidden Layer: **classifier.add(Dense(output_dim=6, init='uniform', activation='relu'))**

The number of inputs isn't defined for this layer as the number of outputs for the previous layer (input layer) has already been defined. The weights and activation function of this layer is defined to be same as the input layer.

c. Output Layer: **classifier.add(Dense(output_dim=1, init='uniform', activation='sigmoid'))**

Since there are only two possible outputs for churn modeling, a single output can be used to represent them. Sigmoid is chosen as the activation function as it provides an output that can be interpreted as a probability. A higher value represents higher probability for customer cancellation.

3. **classifier.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])**

classifier.compile() compiles the given neural network with configurations for loss function, optimizer and metrics for performance. For this model, adam, a highly efficient stochastic gradient descent algorithm, [32] is chosen as the optimizer. Since the activation function for the output layer is a sigmoid function, the output ranges from 0 to 1. The possible outputs are 0 and 1. Thus, binary cross entropy(Equation 2.24) is used as the cost function. Finally, the accuracy is chosen as the criteria for evaluation of the neural network.

4. `classifier.fit(X_train, Y_train, batch_size=10, nb_epoch=10)`

`classifier.fit()` trains the weights of the neural network to fit the training data. The weights are updated after every ten training set samples (`batch_size=10`). The neural network is also trained for the given training set ten times (`nb_epoch=10`).

5. `Y_pred = classifier.predict(X_test)`

`Y_pred = (Y_pred>0.5)`

`classifier.predict()` predicts the output using the test data. For this neural network, a probability greater than 50% is rounded up to 100% and less than 50% is rounded down to 0%.

6. `cm = confusion_matrix(Y_test, Y_pred)`

`confusion_matrix()` generates confusion matrix that compares the output prediction with the correct output from the test data. For example, if

$$cm = \begin{pmatrix} 58 & 7 \\ 14 & 26 \end{pmatrix}$$

This means the model made 84 correct predictions and 21 incorrect ones. So, higher numbers in the diagonal of the confusion matrix means better model.

3.2.4 Results

As seen in Figure 3.5, the neural network manages 83.13% accuracy for the training set and 84.1% for the test set.

```
7990/8000 [=====>.] - ETA: 0s - loss: 0.4113 - acc: 0.8313
8000/8000 [=====] - 100s 12ms/step - loss: 0.4111 - acc: 0.8314
>>> print(cm)
[[1563   32]
 [ 286  119]]
```

Figure 3.5: Output of the ANN code

3.2.5 Optimizing and tuning the ANN

Other than making the neural network deeper (adding more layers), the classifier can be optimized by tuning its hyperparameters. Hyperparameters are the parameters of the network that aren't changed during the training. Examples of hyperparameters are loss function, optimizer, number of epochs, batch size, activation functions, etc.

Another technique for optimization is called dropout. This is generally used to prevent overfitting of the neural network. In this method, some of the neurons are randomly dropped out during some of the training iterations. Due to this, it can decrease or remove the over-reliance of the classifier on some of the neurons which can cause overfitting of the model.

3.3 Convolutional Neural Network (Image Classification)

3.3.1 Image Classification

Image classification is defined as the process of extracting information from images in order to divide them into distinct classes. There are two types of classifications, supervised and unsupervised. Image classification is a complex process, the accuracy of which is mainly related to the characteristics of the dataset, complexity of the problem under analysis, and the robustness of the classification algorithm, [27].

3.3.2 Dataset

For this thesis, image classification is done for a curb-edge dataset. The dataset consists of 272 images which has been divided into training and test dataset at the ratio of 25:9. The images were collected around Idaho State University (ISU) main campus and Measurement and Control Engineering Research Center (MCERC). The images are divided by folders. This also serves as classification for Keras. For example, all the images in the folder `training_set/Curbs` are used as training data classified as curbs by the code. Edges and test sets are arranged in a similar manner.



(a)



(b)

Figure 3.6: Image example for (a) Curb and (b) Edge

Figure 3.6 shows examples of images in training and test sets.

3.3.3 CNN implementation in Keras

The CNN code was written in Anaconda using Spyder IDE in Python 3.6.7. It was executed in both Spyder and Pyscripter IDE.

```
# Importing the Keras libraries and packages
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
from keras.layers import Dense

# Initialising the CNN
classifier = Sequential()

# Step 1 - Convolution
classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))

# Step 2 - Pooling
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Adding a second convolutional layer
classifier.add(Conv2D(32, (3, 3), activation = 'relu'))
classifier.add(MaxPooling2D(pool_size = (2, 2)))

# Step 3 - Flattening
classifier.add(Flatten())

# Step 4 - Full connection
classifier.add(Dense(output_dim = 128, activation = 'relu'))
classifier.add(Dense(output_dim = 1, activation = 'sigmoid'))

# Compiling the CNN
classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

Figure 3.7: Building a CNN in Keras


```

# Fitting the CNN to the images

from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255,
                                    shear_range = 0.2,
                                    zoom_range = 0.2,
                                    horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale = 1./255)

training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                target_size = (64, 64),
                                                batch_size = 10,
                                                class_mode = 'binary')

test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 4,
                                            class_mode = 'binary')

classifier.fit_generator(training_set,
                        steps_per_epoch = 200,
                        epochs = 5,
                        validation_data = test_set,
                        validation_steps = 72)

```

Figure 3.8: Training a CNN in Keras

Figure 3.7 shows the code for building a CNN in Keras and Figure 3.8 shows the code for training the CNN.

1. **classifier = Sequential()**

This defines a sequence of layers named classifier. The layers that are defined after this function call are added sequentially to the neural network.

2. **classifier.add()** adds layer to the neural network.

a. Convolution Layer 1: **classifier.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = 'relu'))**

The CNN is started with a convolution layer with a convolution kernel of size 3×3 (Conv2D(32, (3, 3))). The depth of the kernel is three for RGB image inputs. The number of convolution kernels or feature detectors is initialized at 32. The CNN is also trained with higher

(and lower) number of feature detectors to check for optimization. The convolution layer expects a 64 by 64 RGB image as input. ReLU is the activation function of the layer. It is considered the best activation function for image classification tasks in CNNs.

b. Max Pooling Layer 1: **classifier.add(MaxPooling2D(pool_size = (2, 2)))**

The output of the convolution layer is connected to a max pooling layer of size 2×2 .

c. Convolution and Max Pooling Layer 2: **classifier.add(Conv2D(32, (3, 3), activation = 'relu'))**
classifier.add(MaxPooling2D(pool_size = (2, 2)))

A second convolution and max pooling layer is added so the CNN may be able to identify complex patterns using the combination of convolution layers, i.e. increasing the depth of the deep CNN.

d. Flattening Layer: **classifier.add(Flatten())**

The output of the preceding layers is flattened to form a column matrix for input into the fully connected layer.

e. Fully connected layer: **classifier.add(Dense(output_dim = 128, activation = 'relu'))**

classifier.add(Dense(output_dim = 1, activation = 'sigmoid'))

This creates a fully connected layer consisting of two layers. The hidden layer has 128 neurons and ReLU activation function. Number of neurons for hidden layer is a hyperparameter which can be optimized with trial and error. There is no hard and fast rule for choosing the size of hidden layer. The output layer has one output using a sigmoid activation function.

f. **classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])**

The CNN is compiled using the same hyperparameters as the previous example. It uses “adam”[32] optimizer, “binary cross entropy” loss function and “accuracy” as the metric of evaluation.

g. classifier.fit_generator(training_set, steps_per_epoch = 200, epochs = 5, validation_data = test_set, validation_steps = 72)

classifier.fit_generator() fits the CNN to the given training set (training_set) and validates it using the test set (validation_steps = 72). The CNN is trained using the training set over three iterations (epochs = 3).

4. RESULT AND DISCUSSION

4.1 Output of the CNN

Table 4.1 shows the results of the training and validation as generated by the Keras code.

Table 4.1: Training and validation results for CNN training

Epoch	loss	Acc	val_loss	val_acc
1	0.5821	0.6775	0.7338	0.6806
2	0.4066	0.8125	1.1449	0.7222
3	0.2746	0.8860	1.2715	0.6944
4	0.1839	0.9305	2.4657	0.6389
5	0.1528	0.9410	1.5821	0.7500

In Table 4.1, “acc” gives the accuracy of the neural network for the training set. “val_acc” gives the accuracy for the test set. Similarly, “loss” is the value of loss function of the neural network for the training set. “val_loss” is the loss function for test set.

The CNN scores an accuracy of 94.10% for training set and 75% for test set. The accuracy for test data decreases when the number of epochs is increased. This may indicate overfitting as the size of the training set is relatively small.

4.2 Testing different configurations of CNN

The CNN was tested with two of its hyperparameters varied. The two hyperparameters were number of feature detectors (convolution kernels) in convolution layer and number of neurons in the hidden layer of fully connected layer (ANN). The accuracy results are summarized in Table 4.2 and 4.3.

Table 4.2: Accuracy comparison for different numbers of feature detectors

Accuracy (%)	16 feature detectors	32 feature detectors	64 feature detectors	128 feature detectors
Training Set	93.35	94.10	96.40	89.85
Test Set	63.89	75	66.67	66.67

Table 4.3: Accuracy comparison for different sized hidden layers

Accuracy(%)	64 hidden layer neurons	128 hidden layer neurons	256 hidden layer neurons
Training Set	94.15	94.10	97.50
Test Set	61.11	75	69.44

From the data in Table 4.2 and 4.3, a CNN with 32 feature detectors and 128 hidden layer neurons can be seen as performing best when both training and test set accuracies are considered. While 64 feature detectors and 256 hidden layers configurations sport higher accuracy for training set, they score lower for the test set. This may indicate overfitting.

4.3 Testing with novel images






The CNN is also tested with some images of curbs and edges that are not included in the training or test set. For the separate set of images as shown in Figure 4.4, it scores an accuracy of 100%.



Figure 4.4: Images not included in the training and test sets (set 1)

Another set of 10 images, shown in Table 4.4, were also collected for the test of the CNN. These images contain partially visible curbs or gently sloping edges or neither curbs nor edges. The CNN still shows good accuracy for the curbs and edges (4 out of 5) but it will always label the images without a curb or edge as one or other as the classifier has been trained to classify every image as either curb or edge.

Table 4.4: Set 2 of images and classification by different configurations of the CNN

Images	32 Feature detectors	64 feature detectors	128 feature detectors
 <p>(1) None</p>	Curb	Edge	Curb
 <p>(2) None</p>	Edge	Edge	Edge
 <p>(3) None</p>	Edge	Edge	Edge
 <p>(4) None</p>	Curb	Curb	Curb
 <p>(5) None</p>	Edge	Edge	Edge


 <p>(6) Partial Curb</p>	Edge	Curb	Edge
 <p>(7) Partial Curb</p>	Curb	Edge	Edge
 <p>(8) Slight Edge</p>	Edge	Edge	Edge
 <p>(9) Slight Edge</p>	Edge	Edge	Edge
 <p>(10) Curb</p>	Curb	Curb	Curb

Table 4.4 shows the images used for classification and the prediction by CNNs with different number of feature detectors. The first column contains the images and the 2nd, 3rd and 4th columns contain the predictions made by CNNs with 32, 64 and 128 feature detectors respectively.

For images (1)-(5), images do not contain any curb or edge. The classifier still classifies the images into one of the two classes. It is to be expected as the classifier has been trained to find either curb or edge in every image. However, there is still some information to be gained from these outputs. In image (1), the crack on the road may be triggering the features the CNN has been trained to detect for curbs. In image (2) and (3), the color gradient from the white line to the darker road may be triggering the features of an edge where the higher plane would generally be lit better than the lower. Similar information cannot be extracted from image (4) and (5). The CNNs with 64 and 128 feature detectors perform similarly for image (1)-(5) with only 64 feature detector CNN being an outlier for image (1).

Images (6)–(10) contain curbs, partially hidden or fully visible, and edges with slight gradient. For this set, CNN with 32 and 64 feature detectors get 4 out of 5 correct, however they err for different images. 32 feature detectors can still be considered better as it predicts a curb correctly for the image with more distinct curb (image (7)) while 64 feature detectors do not. 128 feature detectors CNN fares worse in this set than the other two with two errors.

4.4 Optimizing the CNN

The accuracy of the CNN can be improved by making it deeper, i.e. adding more layers to it. Higher number of convolution layers allows the CNN to detect more complex features. A deeper fully connected layer increases the non-linearity of the neural network which also increases the accuracy of classification. All the techniques that are used for optimization and tuning of ANNs

can be applied to fully connected layer of CNN to increase its accuracy. All said, the most effective method for higher accuracy would be a larger dataset.

5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

The application of ANNs and CNNs is on the rise recently. Due to rise in computation and memory capabilities of computers and availability of a variety of datasets, neural networks are applied in fields like robotics, medicine, computer vision, etc. CNNs are especially good at image classification due to its convolution layers. The convolution layer can learn patterns in an image class and with multiple convolution layers stacked, they can learn complex patterns like “dog head shape”, “animal fur texture”, etc. This research shows that a CNN can be used to detect curbs and images even with a small dataset.

5.2 Future Work

Since the final goal of the research is to develop an autonomous wheelchair, the neural network must also be trained to detect other common obstacles such as human beings, animals, vehicles, etc. For this, the training will require a much larger dataset which will also increase the time required for training of the classifier. Images should also be collected from diverse source. Since the images used for training of this CNN was all captured around Idaho State University, it may cause an unknown bias in the classifier during training such as color of the road or the sidewalk.

Adding more layers to the CNN is a good method to increase the accuracy of the network. However, this increases the amount of time required for training. In order to train the neural network within a reasonable amount of time, hardware acceleration may be required. Pre-trained models such as LeNet[16] and AlexNet[2] can also be used and tuned according to need.

Currently, the code only uses RGB sensor of the Kinect. Future extensions in the classifier may also use the depth sensor of the Kinect. Using the RGB and depth sensor at the same time

wasn't possible in the current code as there was no data available for depth sensor images for training.

6. REFERENCES

- [1] Schmidhuber J. *Deep Learning in neural networks: An overview*, Neural Networks 61(2015), pp. 85-117
- [2] Krizhevsky, Alex & Sutskever, Ilya & E. Hinton, Geoffrey. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. Neural Information Processing Systems. 25. 10.1145/3065386.
- [3] Nguyen, Kien & Fookes, Clinton & Ross, Arun & Sridharan, Sridha. (2017). *Iris Recognition with Off-the-Shelf CNN Features: A Deep Learning Perspective*. IEEE Access. PP. 1-1. 10.1109/ACCESS.2017.2784352.
- [4] Disabled World, *Disability in America Infographic*, URL: <https://www.disabled-world.com/disability/statistics/american-disability.php>, Published: December 2011 (Accessed: May 2019)
- [5] Wheelchair Foundation, *Wheelchair Needs in the world*, URL: <https://www.wheelchairfoundation.org/programs/from-the-heart-schools-program/materials-and-supplies/analysis-of-wheelchair-need/>, (Accessed: May 2019)
- [6] KD Smartchair, *Wheelchair Facts, Numbers and Figures [Infographics]*, URL: <https://kdsmartchair.com/blogs/news/18706123-wheelchair-facts-numbers-and-figures-infographic>, Published: January 2015 (Accessed: June 2019)
- [7] Chen W-Y, Jang Y, Wang J-D, Huang W-N, Chang C-C, Mao H-F, Wang Y-H. *Wheelchair-related accidents: Relationship with wheelchair-using behavior in active community wheelchair users*. Arch Phys Med Rehabil 2011;92:892-8.
- [8] Berkvens, Raf & Rymenants, Wouter & Weyn, Maarten & Sleutel, Simon & Loockx, Willy. (2012). *Autonomous Wheelchair: Concept and Exploration*. 38textendash44.

- [9] Levine, S.P.; Bell, D.A.; Jaros, L.A.; Simpson, R.C.; Koren, Y.; Borenstein, J. *The NavChair Assistive Wheelchair Navigation System*. IEEE Trans. Rehabil. Eng. 2002, 7, 443–451.
- [10] Sharma, V.; Simpson, R.C.; Lopresti, E.F.; Schmeler, M. *Clinical evaluation of semiautonomous smart wheelchair architecture (Drive-Safe System) with visually impaired individuals*. J. Rehabil. Res. Dev. 2012, 49, 35–50. [CrossRef] [PubMed]
- [11] Kim, E. Y. *Wheelchair Navigation System for Disabled and Elderly People*. Sensors 2016, 16, 1806; doi:10.3390/s16111806
- [12] Tyka, M. *What is an artificial neural network? Here's everything you need to know*. URL: <https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/>, Published: January 2019 (Accessed: May 2019)
- [13] Kawaguchi, K. *A Multithreaded software model for backpropagation neural network applications*, University of Texas at El Paso, July 2000
- [14] Mahanta, J. *Introduction to Neural Networks, Advantages and Applications*. URL: <https://towardsdatascience.com/introduction-to-neural-networks-advantages-and-applications-96851bd1a207>, Published July 2017 (Accessed April 2019)
- [15] ChrisLB. *Artificial Neuron Model english.png* (License: CC BY-SA 3.0). URL: https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Activation_Functions, Published: July 2005 (Accessed: March 2019)
- [16] Lecun, Yann & Bottou, Leon & Orr, Genevieve & Müller, Klaus-Robert. (1998). *Efficient BackProp*. 10.1007/3-540-49430-8_2.
- [17] Sharma, A. *Understanding Activation Functions in Neural Networks*, URL: <https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0>, Published: March 2017 (Accessed: May 2019)

- [18] Sharma, S. *Activation Functions in Neural Networks*, URL: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>, Published: September 2017 (Accessed May 2019)
- [19] Polamuri, S. *Difference between Softmax Function and Sigmoid Function*, URL: <http://dataaspirant.com/2017/03/07/difference-between-softmax-function-and-sigmoid-function/>, Published: March 2017 (Accessed: May 2019)
- [20] Gajdos, M. *Fun with Neural Networks in Go*. URL: <http://mlexplore.org/2016/07/27/fun-with-neural-networks-in-go/>, Published: 2018 (Accessed May 2019)
- [21] Yadav, S. *Weight Initialization Techniques in Neural Networks*, URL: <https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78>, Published: November 2018 (Accessed: June 2019)
- [22] Wu, Jianxin. *Introduction to Convolutional Neural Networks*, National Key Lab for Novel Software Technology, Nanjing University, China, May 2017
- [23] Deshpande, A. *A Beginner's Guide To Understanding Convolutional Neural Networks*, URL: <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/>, Published: July 2016 (Accessed: June 2019)
- [24] Keras Official Documentation. *Keras: The Python Deep Learning library*, URL: <https://keras.io/>, Accessed: October 2018
- [25] OpenNI. URL: <https://en.wikipedia.org/wiki/OpenNI>
- [26] OpenCV Official Documentation. *About*, URL: <https://opencv.org/about/>
- [27] ArcGIS Desktop. *What is Image Classification?*, URL: <http://desktop.arcgis.com/en/arcmap/latest/extensions/spatial-analyst/image-classification/what-is-image-classification-.htm>, Accessed: June 2019

[28]Image classification is a complex process: [Spatial Modeling in GIS and R for Earth and Environmental Sciences, 2019](#)

[29] URL: <https://www.kaggle.com/barelydedicated/bank-customer-churn-modeling>

[30] Rentschler AJ, Cooper RA, Fitzgerald SG, et al. *Evaluation of selected electric powered wheelchairs using the ANSI/RESNA standards*. Arch Phys Med Rehabil. 2004;85:611–619.

[31] M300 Corpus HD User's manual. URL: https://permobilus.com/wp-content/uploads/2016/11/UM_US_M300_Corpus_HD-1.pdf , Accessed : June 2019

[32] Kingma, D. P., Ba, J. L., *Adam: A method for stochastic optimization*, International Conference on Learning Representations, San Diego, California, 2015.

[33] Baucom Robotics, *Building an AI to play 2048*, URL: <https://www.baucomrobotics.com/projects/2018/5/28/building-an-ai-to-play-2048>, Published: May 2018 (Accessed: June 2019)

7. APPENDIX

7.1 Data preprocessing for Artificial Neural Network

The dataset for churn modeling is preprocessed using the Scikit-learn and Pandas libraries. Pandas is used to read data from the .csv file containing the data from the bank customer. Scikit-learn is used to encode labels and scale the higher valued parameters into standard magnitudes.

```
#Importing the libraries
import numpy as np
import pandas as pd

#Importing the dataset
dataset=pd.read_csv('Churn_Modelling.csv')
X=dataset.iloc[:, 3:13].values
Y=dataset.iloc[:, 13].values

#Encoding categorical data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X_1=LabelEncoder()
X[:, 1]=labelencoder_X_1.fit_transform(X[:,1])
labelencoder_X_2=LabelEncoder()
X[:, 2]=labelencoder_X_2.fit_transform(X[:,2])
onehotencoder=OneHotEncoder(categorical_features=[1])
X=onehotencoder.fit_transform(X).toarray()
X=X[:,1:]

#Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size= 0.2, random_state =0)

#Feature Scaling
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test)
```

7.2 Image pre-processing for Convolution Neural Network

The images are preprocessed for the CNN using Keras' ImageDataGenerator API. The pixel data of all the images (training and test set) are scaled by 1/255. This creates uniformity between all the images and a few images aren't able to influence the neural network more than the others.

In order to create some variety in the training set, some filters are applied randomly. The filters used are shearing, zoom and horizontal flip. All the input images are set to size 64×64 as

the convolution layer expects an input of that size. Larger sizes like 128×128 or 256×256 can be used to gain more information from the image. But using larger sizes will significantly increase the training time of the classifier.

```
from keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip=True)

test_datagen = ImageDataGenerator(rescale = 1./255)

training_set = train_datagen.flow_from_directory('dataset/training_set',
                                                target_size = (64, 64),
                                                batch_size = 16,
                                                class_mode = 'binary')

test_set = test_datagen.flow_from_directory('dataset/test_set',
                                            target_size = (64, 64),
                                            batch_size = 4,
                                            class_mode = 'binary')
```

7.3 Single Prediction in Convolutional Neural Network using Keras

```
import numpy as np
from keras.preprocessing import image
test_image = image.load_img('dataset/single_prediction/curboredge5.jpg', target_size = (64, 64))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis = 0)
result = classifier.predict(test_image)
training_set.class_indices
if result[0][0] == 1:
    prediction = 'edge'
else:
    prediction = 'curb'

print(prediction)
```

7.4 Image Capture using Kinect

The code is interfaced with Kinect using OpenNI2. Since, the image received from the Kinect is in BGR format, OpenCV is used to convert the image to RGB format. Image from Kinect is captured every two seconds.

```

'''
Created on 19Jun2015
Edited on 26Feb2019
Stream rgb video using openni2 opencv-python (cv2)

@Original author: Carlos Torres <carlitos408@gmail.com>
edited for use by: Anupama Shree Dhamala <dhamanup@isu.edu>
'''

import numpy as np
import cv2
import time
from openni import openni2
from openni import _openni2 as c_api

## Initialize openni and check
openni2.initialize() #
if (openni2.is_initialized()):
    print ("openNI2 initialized")
else:
    print ("openNI2 not initialized")

## Register the device
dev = openni2.Device.open_any()

## Create the streams stream
rgb_stream = dev.create_color_stream()

## Check and configure the depth_stream -- set automatically based on bus speed
print ('The rgb video mode is', rgb_stream.get_video_mode()) # Checks rgb video configuration
## Start the streams
rgb_stream.start()

def get_rgb():
    """
    Returns numpy 3L ndarray to represent the rgb image.
    """
    bgr = np.fromstring(rgb_stream.read_frame().get_buffer_as_uint8(), dtype=np.uint8).reshape(480,640,3)
    rgb = cv2.cvtColor(bgr, cv2.COLOR_BGR2RGB)
    return rgb

## main loop
done = False
s = 0
#NEWADDITION
while not done:

    ## Streams
    #RGB
    rgb = get_rgb()

    ## Display the stream syde-by-side
    cv2.imshow('rgb', rgb)

    print(s)
    cv2.imwrite("ex2_"+str(s)+'.png', rgb)
    s+=1 # uncomment for multiple captures
    key = cv2.waitKey(1) & 255
    ## Read keystrokes
    if key == 27: # terminate
        print ("\tESC key detected!")
        done = True

    time.sleep(0.5)
# end while

## Release resources
cv2.destroyAllWindows()
rgb_stream.stop()
openni2.unload()
print ("Terminated")

```