A PROJECT REPORT ON

IMPLEMENATATION OF SELF DRIVING CAR MODEL USING DEEP LEARNING

A Report Submitted in partial fulfillment of the

Academic requirements for the award of the degree of

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

Submitted by

K. SAKETH (1602-15-735-095)

K. SHASHIDHAR (1602-15-735-096)

M. VENKATESH (1602-15-735-113)

Under the esteemed guidance of

Mr. B. UMA MAHESH BABU

Assistant Professor – Dept. of ECE

Vasavi College of Engineering (Autonomous)

Department of Electronics and Communication Engineering

Vasavi College of Engineering

Ibrahimbagh,

Hyderabad - 500031, India

2019

VASAVI COLLEGE OF ENGINEERING

(Autonomous)
Ibrahimbagh, Hyderabad – 500031. (T.S.)

CERTIFICATE

This is to certify that the Project work entitled **"IMPLEMENTATION OF SELF DRIVING CAR MODEL USING DEEP LEARNING"** submitted by **K. SAKETH, K. SHASHIDHAR, M. VENKATESH** in partial fulfillment of the requirement for the award of degree of **BACHELOR OF ENGINEERING** in **ELECTRONICS AND COMMUNICATION ENGINEERING** at Vasavi College Of Engineering, Ibrahimbagh, Hyderabad - 500031, India.

The results obtained in this project have not been submitted to any other university or institution for the award of any degree or diploma.

Mr. M. UMA MAHESH BABU
Project Guide &Asst. Professor
Department of ECE

Dr. E. SREENIVASA RAO
Head of the Department
Department of ECE

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without mentioning of the people whose constant guidance and encouragement made it possible. We take pleasure in presenting before you our project which is result of constant research and implementation of the knowledge gained. We wish to express my sincere gratitude towards **Dr. S.V. RAMANA,** Principal and **Dr. E. SREENIVASA RAO**, Professor and HOD of Electronics and Communication Engineering, Vasavi College of Engineering, for providing us an opportunity to carry out project work on **"IMPLEMENTATION OF SELF DRIVING CAR MODEL USING DEEP LEARNING".**

We would like to gratefully acknowledge the enthusiastic supervision of our Project Guide, **Mr. B. UMA MAHESH BABU**, Assistant Professor of Electronics and Communication Engineering Department at Vasavi College of Engineering, for his constant guidance and encouragement helped us in the timely completion of this project.

We owe our deep gratitude to our project coordinators **Mrs. A. SRILAKSHMI** and **Dr. N. SIVA SANKARA REDDY**, who took keen interest on our project work and guided us all along, till the completion of our project work by providing all the necessary information for developing a good system.

Finally, we thank all the people who have directly or indirectly supported and helped us in completing the project work.

ABSTRACT

A self-driving car is a vehicle that is capable of sensing its environment and navigating without human input. Such vehicles combine a variety of techniques and sensors to identify appropriate navigation paths, obstacles and road signs. The motivation of this project is to build a self-driving car model that is trained to steer by capturing images from the camera attached in the front. And also predict the shortest path to the destination and travel in that path in a network of roads.

Convolutional Neural Network (CNN) is a class of deep, feed-forward artificial neural networks, most commonly applied to analyzing visual imagery. CNNs use relatively little pre-processing compared to other image classification algorithms. It can learn edges, shapes, patterns, etc in the given training set. In our case, a CNN will be used to detect the driving lane and steer the car accordingly. Another CNN can be trained and used to identify traffic lights and road signs but due to poor real-time detection performance, CNNs are not being preferred.

Dijkstra's algorithm is a popular search algorithm, implemented on graphs. This takes a greedy approach and calculates the shortest distance between 2 nodes in a graph. This same algorithm can be used to find out the shortest route to the destination. If the map is created in the form of a graph, this algorithm can be employed and the vehicle can be directed to use the shortest route.

TABLE OF CONTENTS:

List Of Figures:

1. INTRODUCTION

1.1 LEVELS OF AUTOMATION

There are 5 different levels of driving automation where we currently stand with this rapidly advancing technology.

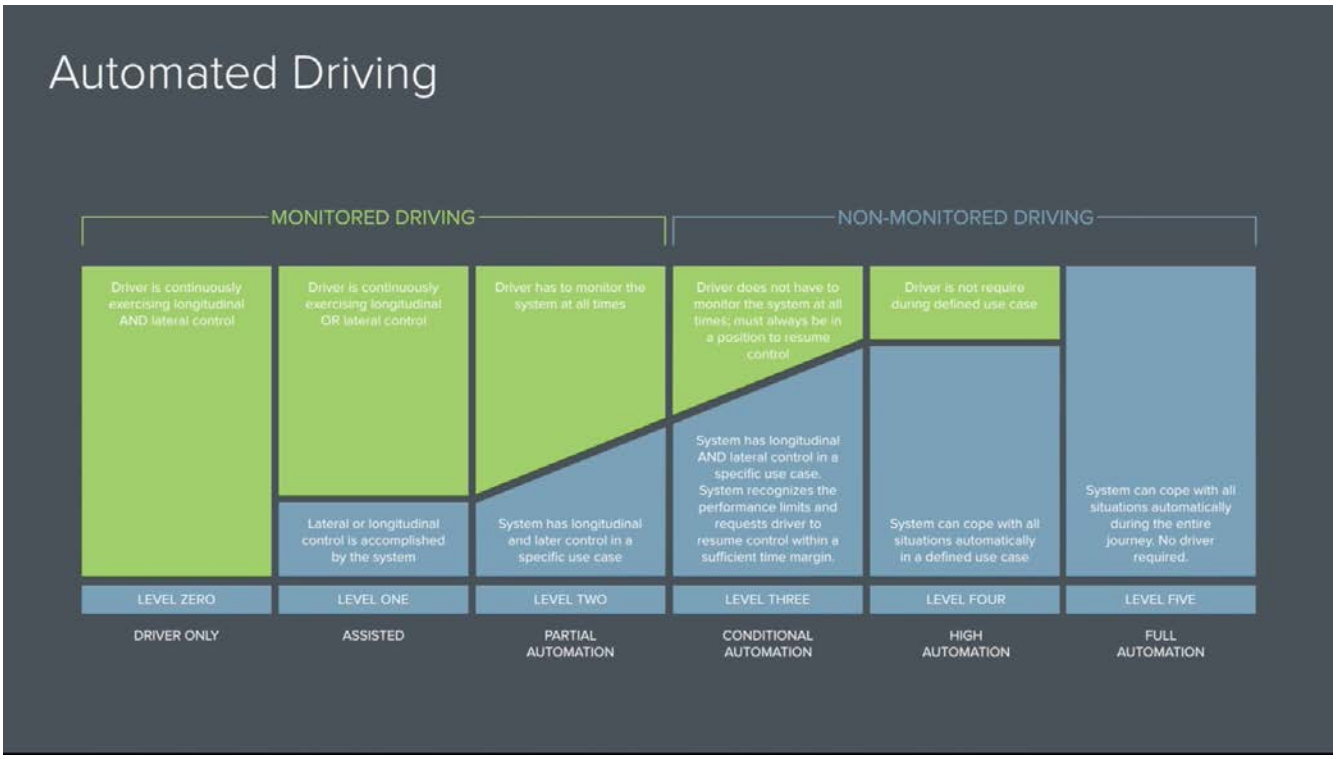


Fig.1.1 Levels of Automation

1.1.0 Level Zero—No Automation

At Level 0 Autonomy, the driver performs all operating tasks like steering, braking, accelerating or slowing down, and so forth.

1.1.1 Level One—Driver Assistance

At this level, the vehicle can assist with some functions, but the driver still handles all accelerating, braking, and monitoring of the surrounding environment. Think of a car that brakes a little extra for you when you get too close to another car on the highway.

1.1.2 Level Two—Partial Automation

Most automakers are currently developing vehicles at this level, where the vehicle can assist with steering or acceleration functions and allow the driver to disengage from some of their tasks. The driver must always be ready to take control of the vehicle and it still responsible for most safety-critical functions and all monitoring of the environment.

1.1.3 Level Three—Conditional Automation

The biggest leap from Level 2 to Levels 3 and above is that starting at Level 3, the vehicle itself controls all monitoring of the environment (using sensors like LiDAR). The driver's attention is still critical at this level, but can disengage from "safety critical" functions like braking and leave it to the technology when conditions are safe. Many current Level 3 vehicles require no human attention to the road at speeds under 37 miles per hour.

1.1.4 Level Four—High Automation

At Levels 4 and 5, the vehicle is capable of steering, braking, accelerating, monitoring the vehicle and roadway as well as responding to events, determining when to change lanes, turn, and use signals.

At Level 4, the autonomous driving system would first notify the driver when conditions are safe, and only then does the driver switch the vehicle into this mode. It cannot determine between more dynamic driving situations like traffic jams or a merge onto the highway.

1.1.5 Level Five—Complete Automation

Last and least (in terms of human involvement), is Level 5 autonomy. This level of autonomous driving requires absolutely no human attention. There is no need for pedals, brakes, or a steering wheel, as the autonomous vehicle system controls all critical tasks, monitoring of the environment and identification of unique driving conditions like traffic jams.

1.2 Current trends

Autonomous cars combine a variety of sensors to perceive their surroundings, such as radar, Lidar, sonar, GPS, odometry and inertial measurement units. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage. Long distance trucks are seen as being in the forefront of adopting and implementing the technology. Driverless vehicles require some form of machine vision for the purpose of visual object recognition. Automated cars are being developed with deep neural networks, a type of deep learning architecture with many computational stages, or levels, in which neurons are simulated from the environment that activate the network. The neural network depends on an extensive amount of data extracted from real-life driving scenarios, enabling the neural network to "learn" how to execute the best course of action.

In May 2018, researchers from MIT announced that they had built an automated car that can navigate unmapped roads. Researchers at their Computer Science and Artificial Intelligence Laboratory (CSAIL) have developed a new system, called MapLite, which allows self-driving cars to drive on roads that they have never been on before, without using 3D maps. The system combines the GPS position of the vehicle, a "sparse topological map" such as OpenStreetMap, (i.e. having 2D features of the roads only), and a series of sensors that observe the road conditions.

One way to assess the progress of automated vehicles is to compute the average distance driven between "disengagements", when the automated system is turned off, typically by a human driver. In 2017, Waymo reported 63 disengagements over 352,545 miles (567,366 km) of testing, or 5,596 miles (9,006 km) on average, the highest among companies reporting such figures. Waymo also traveled more distance in total than any other. Their 2017 rate of 0.18 disengagements per 1,000 miles (1,600 km) was an improvement from 0.2 disengagements per 1,000 miles (1,600 km) in 2016 and 0.8 in 2015. In March 2017, Uber reported an average of 0.67 miles (1.08 km) per disengagement. In the final three months of 2017, Cruise Automation (now owned by GM) averaged 5,224 miles (8,407 km) per disruption over 62,689 miles (100,888 km).

2. LITERATURE SURVEY

2.1 MACHINE LEARNING BASICS

Machine Learning mainly divided into three categories, which are as follows-

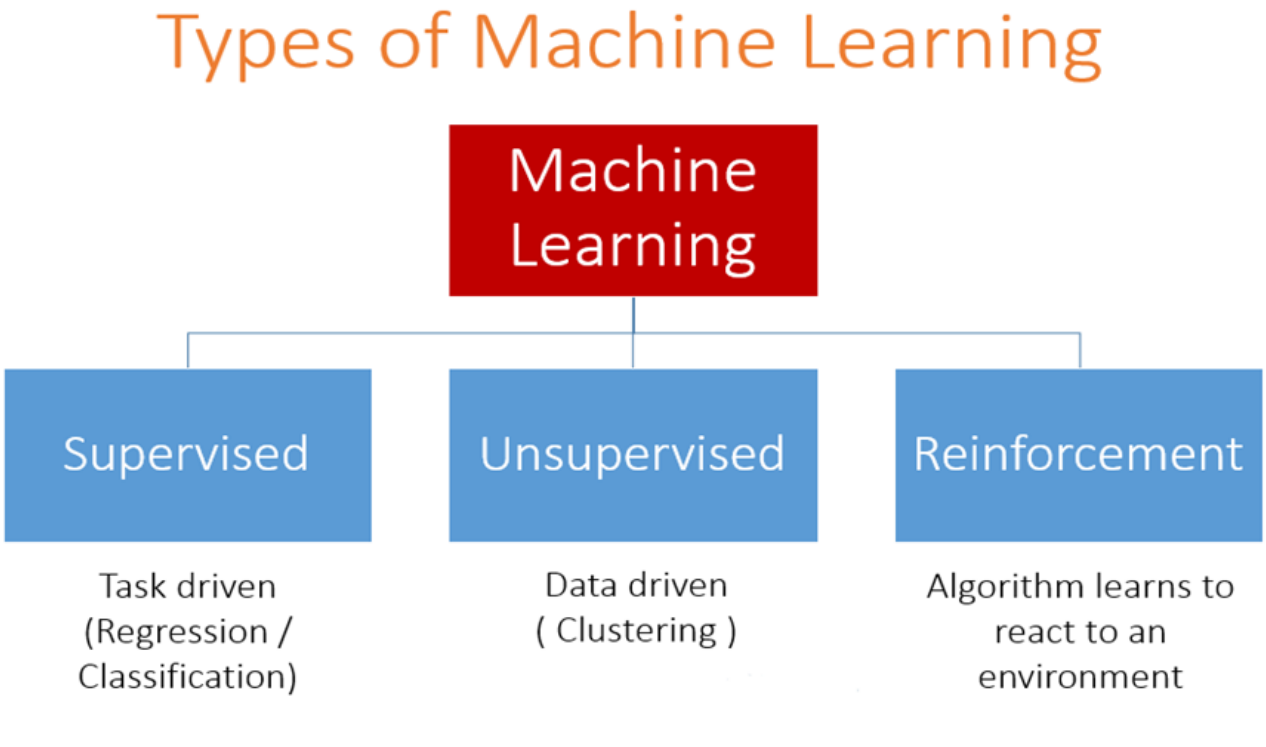


Fig.2.1 Types of Machine Learning

2.1.1 Supervised Learning

Supervised Learning is the first type of machine learning, in which labelled data used to train the algorithms. In supervised learning, algorithms are trained using marked data, where the input and the output are known. We input the data in the learning algorithm as a set of inputs, which is called as Features, denoted by X along with the corresponding outputs, which is indicated by Y, and the algorithm learns by comparing its actual production with correct outputs to find errors. It then modifies the model accordingly. The raw data divided into two parts. The first part is for training the algorithm, and the other region used for test the trained algorithm.

Supervised learning uses the data patterns to predict the values of additional data for the labels. This method will commonly use in applications where historical data predict likely upcoming events.
The Supervised Learning mainly divided into two parts which are as follows-

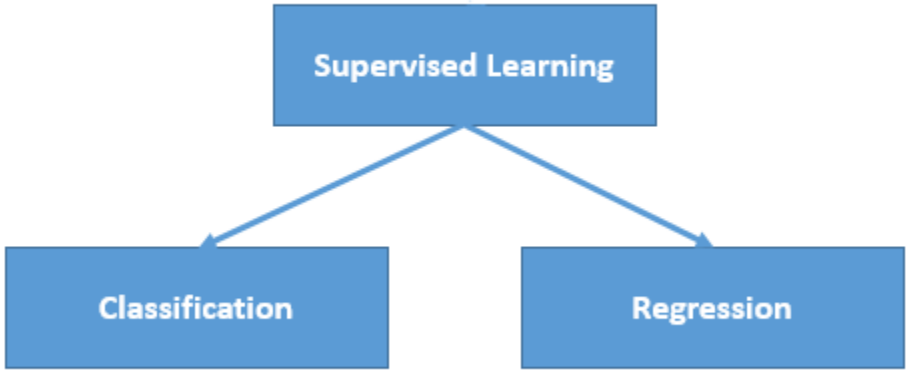


Fig.2.2 Types of Supervised Learning

2.1.1.a *Regression*

Regression is the type of Supervised Learning in which labelled data used, and this data is used to make predictions in a continuous form. The output of the input is always ongoing, and the graph is linear. Regression is a form of predictive modelling technique which investigates the relationship between a dependent variable[Outputs] and independent variable[Inputs]. This technique used for forecasting the weather, time series modelling, process optimisation. Ex:- One of the examples of the regression technique is House Price Prediction, where the price of the house will predict from the inputs such as No of rooms, Locality, Ease of transport, Age of house, Area of a home.

2.1.1.b Classification

Classification is the type of Supervised Learning in which labelled data can use, and this data is used to make predictions in a non-continuous form. The output of the information is not always continuous, and the graph is non-linear. In the classification technique, the algorithm learns from the data input given to it and then uses this learning to classify new observation. This data set may merely be bi-class, or it may be multi-class too. Ex:- One of the examples of classification problems is to check whether the email is spam or not spam by train the algorithm for different spam words or emails.

2.1.2 Unsupervised Learning

Unsupervised Learning is the second type of machine learning, in which unlabeled data are used to train the algorithm, which means it used against data that has no historical labels. What is being showing must figure out by the algorithm. The purpose is to explore the data and find some structure within. In unsupervised learning the data is unlabeled, and the input of raw information directly to the algorithm without pre-processing of the data and without knowing the output of the data and the data can not divide into a train or test data. The algorithm figures out the data and according to the data segments, it makes clusters of data with new labels.

2.1.3 Reinforcement Learning

Reinforcement Learning is the third type of machine learning in which no raw data is given as input instead reinforcement learning algorithm have to figures out the situation on their own. The reinforcement learning frequently used for robotics, gaming, and navigation. With reinforcement learning, the algorithm discovers through trial and error which actions yield the most significant rewards. This type of training has three main components which are the agent which can describe as the learner or decision maker, the environment which described as everything the agent interacts with and actions which represented as what the agent can do.

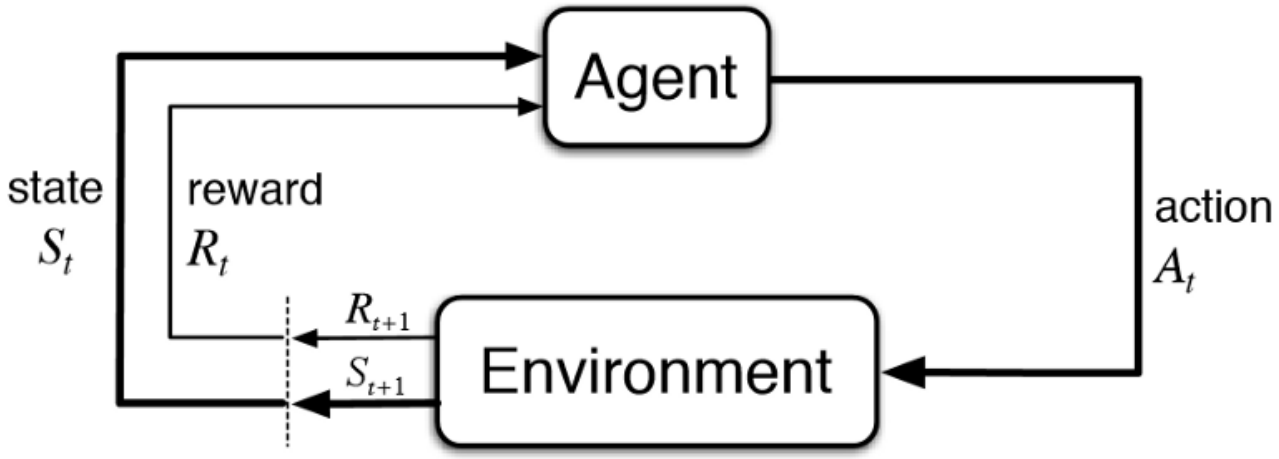


Fig.2.3 Process of Reinforcement Learning

The objective is for the agent to take actions that maximise the expected reward over a given measure of time. The agent will reach the goal much quicker by following a good policy. So the purpose of reinforcement learning is to learn the best plan.

2.2 INTRODUCTION TO DEEP LEARNING

Deep learning is a machine learning method that takes in an input X, and uses it to predict an output of Y. In Figure, we represent linear regression with a neural network diagram. The diagram shows the connectivity among the inputs and output, but does not depict the weights or biases (which are given implicitly).

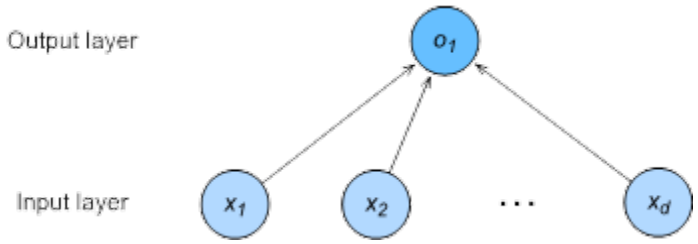


Fig.2.4 Simple Representation of a Neural Networks

In the above network, the inputs are x1,x2,…xd. Sometimes the number of inputs are referred to as the feature dimension. For linear regression models, we act upon dd inputs and output 11 value. Because there is just a single computed neuron (node) in the graph, we can think of linear models as neural networks consisting of just a single neuron. Since all inputs are connected to all outputs (in this case it's just one), this layer can also be regarded as an instance of a *fully-connected layer*, also commonly called a *dense layer*.

Neural networks derive their name from their inspirations in neuroscience. Although linear regression predates computation neuroscience, many of the models we subsequently discuss truly owe to neural inspiration. To understand the neural inspiration for artificial neural networks it is worth while considering the basic structure of a neuron. For the purpose of the analogy it is sufficient to consider the *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals) which connect to other neurons via *synapses*.

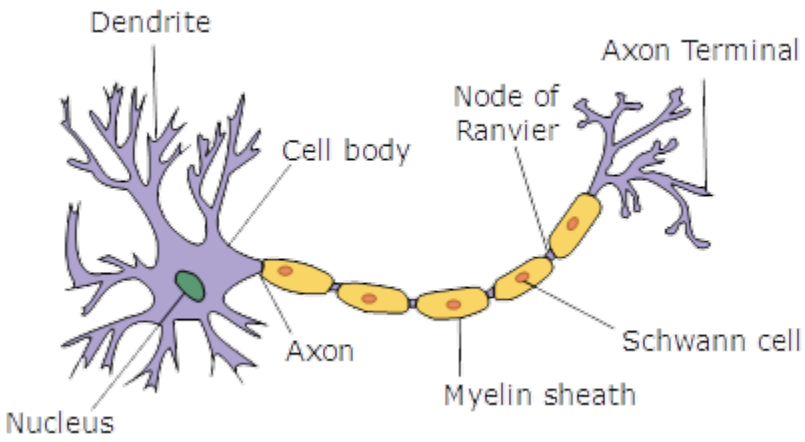


Fig.2.5 Biological Representation of a Neuron

Information Xi arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites. In particular, that information is weighted by *synaptic weights* Wi which determine how to respond to the inputs (e.g. activation or inhibition via XiWi). All this is aggregated in the nucleus $y=\sum_i Xi+$ b, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via σ(y). From there it either reaches its destination (e.g. a muscle) or is fed into another neuron via its dendrites.

The simplest deep networks are called multilayer perceptrons, and they consist of many layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence).

Modern deep learning provides a powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large models and sufficiently large datasets of labeled training examples. Other tasks, that cannot be described as associating one vector to another, or that are difficult enough that a person would require time to think and reflect in order to accomplish the task, remain beyond the scope of deep learning for now.

2.3 HIDDEN LAYERS

The way to create a Deep Neural Network is to stack many layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a *multilayer perceptron*, often abbriviated as *MLP*. The neural network diagram for an MLP looks like this:

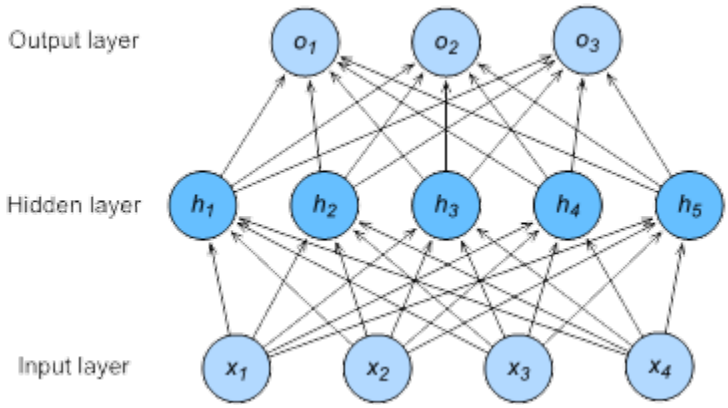


Fig.2.6 Multilayer perceptron with One Hidden Layer

The multilayer perceptron above has 4 inputs and 3 outputs, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, building this network would consist of implementing 2 layers of computation. The neurons in the input layer are fully connected to the inputs in the hidden layer. Likewise, the neurons in the hidden layer are fully connected to the neurons in the output layer.

2.4 ACTIVATION FUNCTIONS

2.4.1 Rectified Linear Unit (ReLU)

ReLU(Rectified Linear Unit)s provide a very simple nonlinear transformation. Given the element z, the function is defined as the maximum of that element and 0.

ReLU(z)=max(z,0).

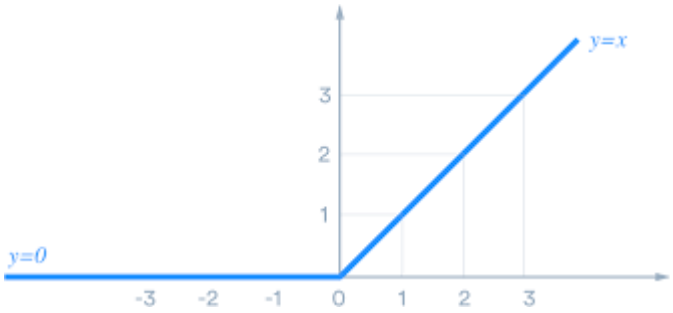


Fig.2.7 ReLU Activation Plot

2.4.2 Sigmoid

The sigmoid function transforms its inputs which take values in RR to the interval (0,1)(0,1). For that reason, the sigmoid is often called a *squashing* function: it squashes any input in the range (-inf, inf) to some value in the range (0,1).

$$\text{sigmoid}(x) = 11 + e^{-x}$$

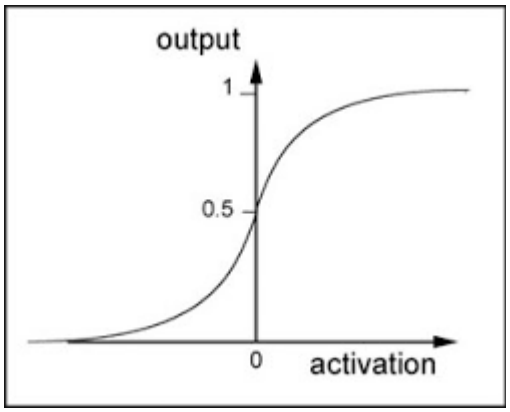


Fig.2.8 Sigmoid Activation

2.4.3 Hyperbolic Tangent (Tanh)

Like the sigmoid function, the tanh (Hyperbolic Tangent) function also squashes its inputs, transforms them into elements on the interval between -1 and 1:

$$\tanh(x) = 1 - e^{-2x}1 + e^{-2x}$$

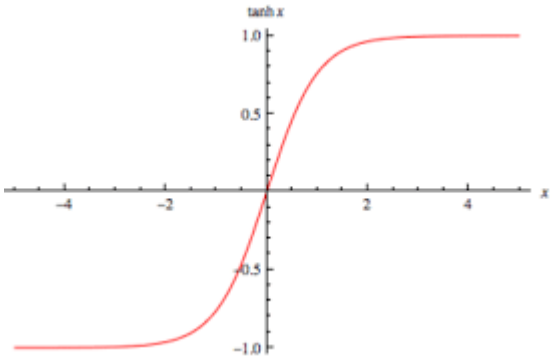


Fig.2.9 Tanh Activation

2.4.4 Exponential Linear Unit (ELU)

Less widely-used modification of ReLU, which is said to lead to higher classification results than traditional ReLU. It follows the same rule for x>= 0 as ReLU, and increases exponentially for x < 0. ELU tries to make the mean activations closer to zero which speeds up training.

It has just one parameter alpha, which controls the scale of the negative part, and by default is set to 1.0.

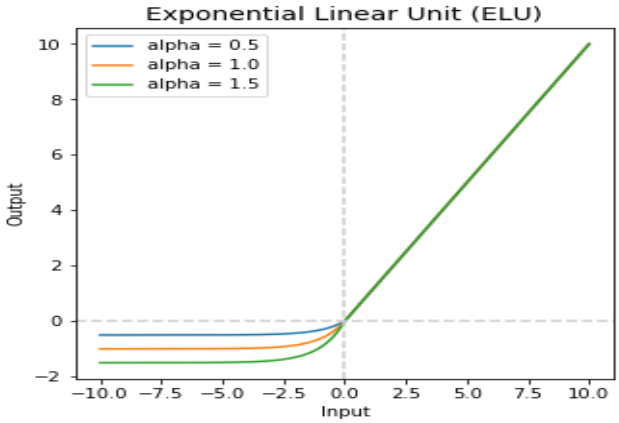


Fig.2.10 Exponential Linear unit (ELU) activation

2.5 FORWARD PROPAGATION

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following, we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call backward.

For the sake of simplicity, let's assume that the input example is$x \in \mathbb{R}^d$and there is no bias term. Here the intermediate variable is:

$$z = W^{(1)}x$$

$W^{(1)} \in \mathbb{R}^{hXd}$is the weight parameter of the hidden layer. After entering the intermediate variable $z \in \mathbb{R}^h$into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h.

$$h=\phi(z)$$

The hidden variable h is also an intermediate variable. Assuming the parameters of the output layer only possess a weight of $W^{(2)} \in \mathbb{R}^{qXh}$, we can obtain an output layer variable with a vector length of q:

$$o = W^{(2)}h$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L=l(o,y)$$

According to the definition of $\ell 2$ norm regularization, given the hyper-parameter λ, the regularization term is

$$s = \frac{\lambda}{2}\left(\left\|W^{(1)}\right\|_F^2 + \left\|W^{(2)}\right\|_F^2\right)$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L2L2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s$$

We refer to J as the objective function

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. The figure below contains the graph associated with the simple network described above. The lower-left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.

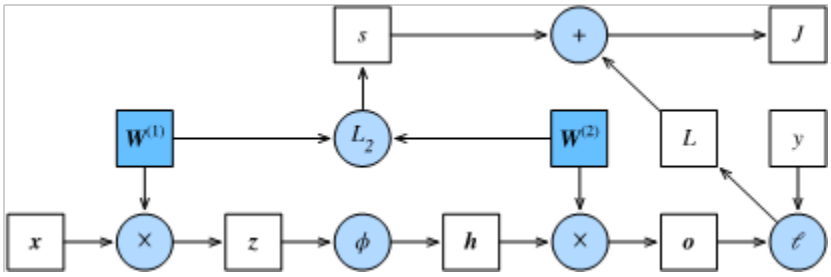


Fig.2.11 Pictorial Representation of Forward Propagation

2.6 BACKPROPAGATION

Backpropagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the 'chain rule' in calculus. Assume that we have functions Y=f(X) and Z=g(Y)=g∘f(X), in which the input and the output X,Y,Z are tensors of arbitrary shapes. By using the chain rule, we can compute the derivative of Z wrt. X via

$$\frac{\partial Z}{\partial X}=\text{prod}\left(\frac{\partial Z}{\partial Y},\frac{\partial Y}{\partial X}\right)$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions have been carried out. For vectors, this is straightforward: it is simply matrix-matrix multiplication and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer are $W^{(1)}$ and $W^{(2)}$. The objective of backpropagation is to calculate the gradients $\partial J/\partial W^{(1)}$ and $\partial J/\partial W^{(2)}$. To accomplish this, we will apply the chain rule and calculate, in turn, the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function J=L+s with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L}=1 \text{ and } \frac{\partial J}{\partial s}=1$$

Next, we compute the gradient of the objective function with respect to variable of the output layer **o** according to the chain rule.

$$\frac{\partial J}{\partial o}=\text{prod}(\frac{\partial J}{\partial L},\frac{\partial L}{\partial o})=\frac{\partial L}{\partial o} \in \mathbb{R}^q$$

Next, we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial W^{(1)}}=\lambda W^{(1)} \text{ and } \frac{\partial s}{\partial W^{(2)}}=\lambda W^{(2)}$$

Now we are able calculate the gradient $\partial J/\partial W^{(2)} \in \mathbb{R}^{qXh}$of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial W^{(2)}}=\text{prod}(\frac{\partial J}{\partial o},\frac{\partial o}{\partial W^{(2)}})+\text{prod}(\frac{\partial J}{\partial s},\frac{\partial s}{\partial W^{(2)}})=\frac{\partial J}{\partial o} h^T+\lambda W^{(2)}$$

To obtain the gradient with respect to $W^{(1)}$ we need to continue backpropagation along the output layer to the hidden layer. The gradient with respect to the hidden layer's outputs$\partial J/\partial h \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial h}= \text{prod}(\frac{\partial J}{\partial o},\frac{\partial o}{\partial h}) = {W^{(2)}}^T \frac{\partial J}{\partial o}$$

Since the activation function ϕ applies element-wise, calculating the gradient$\partial J/\partial z \in \mathbb{R}^h$ of the intermediate variable z requires that we use the element-wise multiplication operator, which we denote by $\odot$.

$$\frac{\partial J}{\partial z}= \text{prod}(\frac{\partial J}{\partial h},\frac{\partial h}{\partial z}) =\frac{\partial J}{\partial h} \odot \phi'(z)$$

Finally, we can obtain the gradient $\partial J/\partial W^{(1)} \in \mathbb{R}^{hXd}$of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial W^{(1)}}=\text{prod}(\frac{\partial J}{\partial z},\frac{\partial z}{\partial W^{(1)}})+\text{prod}(\frac{\partial J}{\partial s},\frac{\partial s}{\partial W^{(1)}})=\frac{\partial J}{\partial z} x^T+\lambda W^{(1)}$$

2.7 OPTIMIZATION ALGORITHMS

When training models, we use optimization algorithms to continue updating the model parameters to reduce the value of the model loss function. When iteration ends, model training ends along with it. The model parameters we get here are the parameters that the model learned through training.

Optimization algorithms are important for deep learning. On the one hand, training a complex deep learning model can take hours, days, or even weeks. The performance of the optimization algorithm directly affects the model's training efficiency. On the other hand, understanding the principles of different optimization algorithms and the meanings of their hyperparameters will enable us to tune the hyperparameters in a targeted manner to improve the performance of deep learning models.

For a deep learning problem, we will usually define a loss function first. Once we have the loss function, we can use an optimization algorithm in attempt to minimize the loss. In optimization, a loss function is often referred to as the objective function of the optimization problem. Traditionally, optimization algorithms usually only consider minimizing the objective function. In fact, any maximization problem can be easily transformed into a minimization problem: we just need to use the opposite of the objective function as the new objective function.

2.7.1 Gradient Descent

The input of the objective function is a vector and the output is a scalar. We assume that the input of the target function $\mathbf{f}:\mathbb{R}^d \rightarrow \mathbb{R}$ is the d-dimensional vector $\mathbf{x}=[x_1, x_2, \ldots\ldots, x_d]^T$. The gradient of the objective function **f(x)** with respect to **x** is a vector consisting of **d** partial derivatives:

$$\nabla_x \mathbf{f(x)} = \left[\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots\ldots, \frac{\partial f(x)}{\partial x_d}\right]^T$$

For brevity, we use **∇f(x)** instead of $\nabla_x \mathbf{f(x)}$. Each partial derivative element $\partial \mathbf{f(x)}/\partial x_i$ in the gradient indicates the rate of change of **f** at **x** with respect to the input x_i. To measure the rate of change of **f** in the direction of the unit vector **u (‖u‖=1**), in multivariate calculus, the directional derivative of **f** at **x** in the direction of **u** is defined as

$$D_u \mathbf{f(x)} = \lim_{h \rightarrow 0} \frac{f(x+hu)-f(x)}{h}$$

According to the property of directional derivatives, the aforementioned directional derivative can be rewritten as

$$D_u \mathbf{f(x)} = \nabla \mathbf{f(x)} \cdot \mathbf{u}$$

The directional derivative $D_u \mathbf{f(x)}$ gives all the possible rates of change for **f** along **x**. In order to minimize f, we hope to find the direction the will allow us to reduce **f** in the fastest way. Therefore, we can use the unit vector **u** to minimize the directional derivative $D_u \mathbf{f(x)}$.

ForD_u**f(x) =‖∇f(x)‖·‖u‖·cos(θ)=‖∇f(x)‖·cos(θ),** Here, θ is the angle between the gradient **∇f(x)** and the unit vector **u**. When **θ=π**, **cos (θ)** gives us the minimum value −1. So when **u** is in a direction that is opposite to the gradient direction **∇f(x)**, the direction derivativeD_u**f(x)** is minimized. Therefore, we may continue to reduce the value of objective function **f** by the gradient descent algorithm:

x←x−η∇f(x).

Similarly,**η** (positive) is called the learning rate.

Now we are going to construct an objective function **f(x)** =$x_1^2 + 2x_2^2$with a two-dimensional vector **x**=$[x_1, x_2]^T$as input and a scalar as the output. So we have the gradient **∇f(x)**=$[2x_1, 4x_2]^T$. We will observe the iterative trajectory of independent variable x by gradient descent from the initial position **[−5,−2]**. First, we are going to define two helper functions. The first helper uses the given independent variable update function to iterate independent variable **x** a total of 20 times from the initial position **[−5,−2]**. The second helper will visualize the iterative trajectory of independent variable **x**.

Next, we observe the iterative trajectory of the independent variable at learning rate **0.1**. After iterating the independent variable **x** 20 times using gradient descent, we can see that. eventually, the value of **x** approaches the optimal solution **[0,0]**.

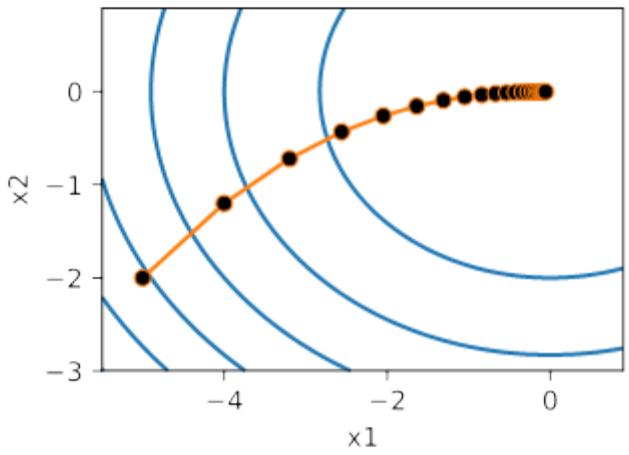


Fig.2.12 Example of a Gradient Descent Optimization

2.7.2 Stochastic Gradient Descent (SGD)

In deep learning, the objective function is usually the average of the loss functions for each example in the training data set. We assume that $f_i(x)$ is the loss function of the training data instance with **n** examples, an index of **i**, and parameter vector of **x**, then we have the objective function

$$\mathbf{f(x)} = \frac{1}{n}\sum_{i=1}^{n} f_i\,(x)$$

The gradient of the objective function at **x** is computed as

$$\nabla\,\mathbf{f(x)} = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x)$$

If gradient descent is used, the computing cost for each independent variable iteration is O(n), which grows linearly with n. Therefore, when the model training data instance is large, the cost of gradient descent for each iteration will be very high.

Stochastic gradient descent (SGD) reduces computational cost at each iteration. At each iteration of stochastic gradient descent, we uniformly sample an index **i∈{1,…,n}** for data instances at random, and compute the gradient$\nabla f_i(x)$ to update **x**:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(x)$$

Here, **η** is the learning rate. We can see that the computing cost for each iteration drops from **O(n)** of the gradient descent to the constant **O(1)**. We should mention that the stochastic gradient$\nabla f_i(x)$ is the unbiased estimate of gradient **∇f(x).**

$$\nabla f_i(x) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(x) = \nabla\,\mathbf{f(x)}$$

This means that, on average, the stochastic gradient is a good estimate of the gradient.

Now, we will compare it to gradient descent by adding random noise with a mean of 0 to the gradient to simulate a SGD.

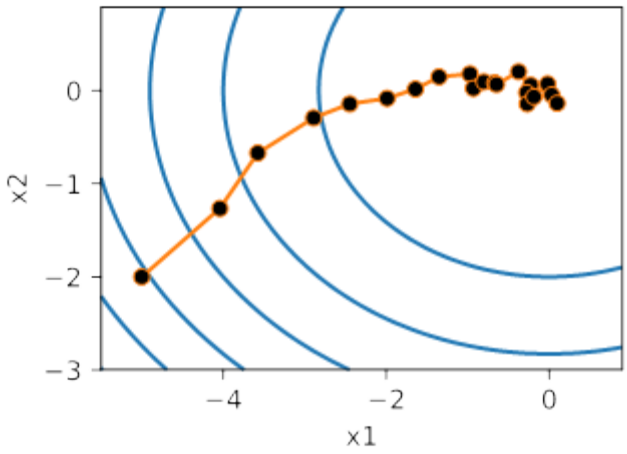


Fig.2.13 Example of a SGD Optimization

As we can see, the iterative trajectory of the independent variable in the SGD is more tortuous than in the gradient descent. This is due to the noise added in the experiment, which reduced the accuracy of the simulated stochastic gradient. In practice, such noise usually comes from individual examples in the training data set.

2.7.3 Momentum

In the **Gradient Descent and Stochastic Gradient Descent** algorithms, the gradient of the objective function's independent variable represents the direction of the objective function's fastest descend at the current position of the independent variable. Therefore, gradient descent is also called steepest descent. In each iteration, the gradient descends according to the current position of the independent variable while updating the latter along the current position of the gradient. However, this can lead to problems if the iterative direction of the independent variable relies exclusively on the current position of the independent variable.

The momentum method was proposed to solve the gradient descent problem described above. Since mini-batch stochastic gradient descent is more general than gradient descent, the subsequent discussion will continue to use the definition for mini-batch stochastic gradient descentg_t at time step **t**. We set the independent variable at time step **t** to$x_t$and the learning rate toη_t. At time step 0, momentum creates the velocity variablev_0 and initializes its elements to zero. At time step t>0, momentum modifies the steps of each iteration as follows:

$$v_t \leftarrow \gamma v_{t-1} + \eta_t g_t$$

$$x_t \leftarrow x_{t-1} - v_t$$

Here, the momentum hyperparameter **γ** satisfies **0≤γ<1**. When **γ=0**, momentum is equivalent to a mini-batch SGD.

2.7.3.1 Exponentially Weighted Moving Average (EWMA)

In order to understand the momentum method mathematically, we must first explain the exponentially weighted moving average (EWMA). Given hyperparameter **0≤γ<1**, the variable y_t of the current time step **t** is the linear combination of variable y_{t-1} from the previous time step **t−1** and another variable x_t of the current step.

$$y_t = \gamma y_{t-1} + (1-\gamma)x_t$$

We can expand y_t:

$$y_t = (1-\gamma)x_t + \gamma y_{t-1}$$

$$= (1-\gamma)x_t + \textbf{(1-}\gamma\textbf{)}\gamma x_{t-1} + \gamma^2 y_{t-2}$$

$$= (1-\gamma)x_t + \textbf{(1-}\gamma\textbf{)}\gamma x_{t-1} + (1-\gamma)\gamma^2 x_{t-2} + \gamma^3 y_{t-3} \ldots\ldots\ldots\ldots$$

Let **n=1/(1−γ),** so $(1-1/n)^n = \gamma^{1/(1-\gamma)}$. Because

$$\lim_{n\to\infty}(1-1/n)^n = exp(-1) \approx 0.3679,$$

when **γ→1,** $\gamma^{1/(1-\gamma)}$ **= exp(−1)**. For example, $0.95^{20} \approx \exp(-1)$. If we treat **exp(−1)** as a relatively small number, we can ignore all the terms that have $\gamma^{1/(1-\gamma)}$ or coefficients of higher order $\gamma^{1/(1-\gamma)}$ in them. For example, when γ=0.95,

$$y_t \approx 0.05 \sum_{i=0}^{19} 0.95^i \, x_{t-i}$$

Therefore, in practice, we often treat y_t as the weighted average of the x_t values from the last 1/(1−γ) time steps. For example, when γ=0.95, y_t can be treated as the weighted average of the x_t values from the last 20 time steps; when γ=0.9, y_t can be treated as the weighted average of the x_t values from the last 10 time steps. Additionally, the closer the x_t value is to the current time step t, the greater the value's weight (closer to 1).

Now, we are going to deform the velocity variable of momentum:

$$v_t \leftarrow \gamma v_{t-1} + (1-\gamma)\left(\frac{\eta_t}{1-\gamma} g_t\right).$$

By the form of EWMA, velocity variable v_t is actually an EWMA of time series $\{\eta_{t-1} g_{t-1}/(1-\gamma)$**:i=0,…,1/(1−γ)−1}**. In other words, considering mini-batch SGD, the update of an independent variable with momentum at each time step approximates the EWMA of the updates in the last 1/(1−γ) time steps without momentum, divided by 1−γ. Thus, with momentum, the movement size at each direction not only depends on the current gradient, but also depends on whether the past gradients are aligned at each direction. In the optimization problem mentioned earlier in this section, all the gradients are positive in the horizontal direction (rightward), but are occasionally positive (up) or negative (down) in the vertical direction. As a result, we can use a larger learning rate to allow the independent variable move faster towards the optimum.

2.7.4 Adam

Adam uses the momentum variable v_t and variable s_t, which is an EWMA on the squares of elements in the mini-batch stochastic gradient from RMSProp, and initializes each element of the variables to 0 at time step 0. Given the hyperparameter $0 \leq \beta_1 < 1$ (the author of the algorithm suggests a value of 0.9), the momentum variable v_t at time step t is the EWMA of the mini-batch stochastic gradient g_t:

$$v_t \leftarrow \beta_1 v_{t-1} + (1-\beta_1) g_t$$

Just as in RMSProp, given the hyperparameter $0 \leq \beta_2 < 1$ (the author of the algorithm suggests a value of 0.999), After taken the squares of elements in the mini-batch stochastic gradient, find $g_t \odot g_t$ and perform EWMA on it to obtain s_t:

$$s_t \leftarrow \beta_2 s_{t-1} + (1-\beta_2) g_t \odot g_t$$

Since we initialized elements in v_0 and s_0 to 0, we get $v_t = (1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} g_i$ at time step t. Sum the mini-batch stochastic gradient weights from each previous time step to get $(1-\beta_1)\sum_{i=1}^{t} \beta_1^{t-i} = 1-\beta_1^t$. Notice that when t is small, the sum of the mini-batch stochastic gradient weights from each previous time step will be small. For example, when $\beta_1 = 0.9$, $v_1 = 0.1 g_1$. To eliminate this effect, for any time step t, we can divide v_t by $1-\beta_1^t$, so that the sum of the mini-batch stochastic gradient weights from each previous time step is 1. This is also called bias correction. In the Adam algorithm, we perform bias corrections for variables v_t and s_t:

$$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_1^t}$$

$$\hat{s}_t \leftarrow \frac{s_t}{1-\beta_2^t}$$

Next, the Adam algorithm will use the bias-corrected variables $\hat{v}_t$ and $\hat{s}_t$ from above to re-adjust the learning rate of each element in the model parameters using element operations.

$$\hat{g}_t \leftarrow \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

Here, η is the learning rate while ϵ is a constant added to maintain numerical stability, such as 10^{-8}. Just as for Adagrad, RMSProp, and Adadelta, each element in the independent variable of the objective function has its own learning rate. Finally, use $\hat{g}_t$ to iterate the independent variable:

$$x_t \leftarrow x_{t-1} - \hat{g}_t$$

2.8 CONVOLUTIONAL NEURAL NETWORKS

Convolutional networks, also known as convolutional neural networks,or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications.

2.8.1 The Cross-Correlation Operator

In a convolutional layer, an input array and a correlation kernel array output an array through a cross-correlation operation. Let's see how this works for two dimensions. As shown below, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. This array is also called a kernel or filter in convolution computations. The shape of the kernel window (also known as the convolution window) depends on the height and width of the kernel, which is 2×2.

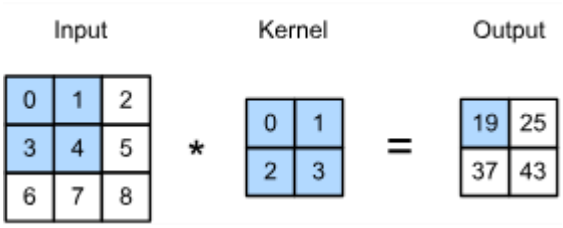


Fig.2.14 Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation:

$$0\times0+1\times1+3\times2+4\times3=19$$

In the two-dimensional cross-correlation operation, the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. The output array has a height of 2 and width of 2, and the four elements are derived from a two-dimensional cross-correlation operation:

$$0\times0+1\times1+3\times2+4\times3=19,$$

$$1\times0+2\times1+4\times2+5\times3=25,$$

$$3\times0+4\times1+6\times2+7\times3=37,$$

$$4\times0+5\times1+7\times2+8\times3=43.$$

Note that the output size is smaller than the input. In particular, the output size is given by the input size $\mathbf{H\times W}$ minus the size of the convolutional kernel $\mathbf{h\times w}$ via $\mathbf{(H-h+1)\times(W-w+1)}$

2.8.2 Convolutional Layers

The convolutional layer cross-correlates the input and kernels and adds a scalar bias to get the output. The model parameters of the convolutional layer include the kernel and scalar bias. When training the model, we usually randomly initialize the kernel and then continuously iterate the kernel and bias.

2.8.3 Padding and Stride

In general, assuming the input shape is nh×nw and the convolution kernel window shape is kh×kw then the output shape will be

$$(nh-kh+1)\times(nw-kw+1)$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window. In several cases we might want to change the dimensionality of the output:

- Multiple layers of convolutions reduce the information available at the boundary, often by much more than what we would want. If we start with a 240x240 pixel image, 10 layers of 5x5 convolutions reduce the image to 200x200 pixels, effectively slicing off 30% of the image and with it obliterating anything interesting on the boundaries. Padding mitigates this problem.
- In some cases we want to reduce the resolution drastically, e.g. halving it if we think that such a high input dimensionality is not required. In this case we might want to subsample the output. Strides address this.
- In some cases we want to increase the resolution, e.g. for image superresolution or for audio generation. Again, strides come to our rescue.
- In some cases we want to increase the length gently to a given size (mostly for sentences of variable length or for filling in patches). Padding addresses this.

2.8.3.a Padding

As we saw so far, convolutions are quite useful. Alas, on the boundaries we encounter the problem that we keep on losing pixels. For any given convolution it's only a few pixels but this adds up as we discussed above. If the image was larger things would be easier - we could simply record one that's larger. Unfortunately, that's not what we get in reality. One solution to this problem is to add extra pixels around the boundary of the image, thus increasing the effective size of the image (the extra pixels typically assume the value 0). In the figure below we pad the 3×5 to increase to 5×7. The corresponding output then increases to a 4×6 matrix.

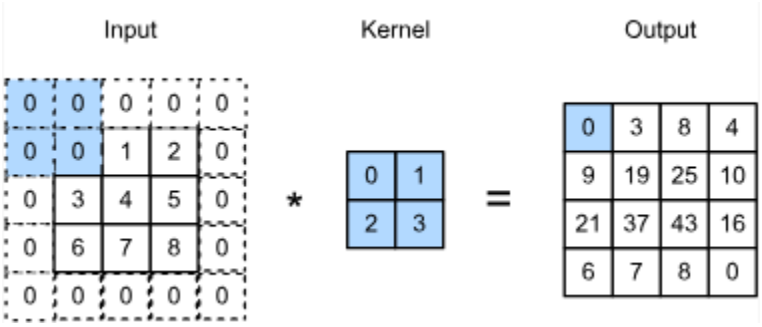


Fig.2.15 Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: 0×0+0×1+0×2+0×3=0

In general, if a total of ph rows are padded on both sides of the height and a total of pw columns are padded on both sides of width, the output shape will be

$$(nh-kh+ph+1)\times(nw-kw+pw+1),$$

This means that the height and width of the output will increase by ph and pw respectively.

2.8.3.b Stride

When computing the cross-correlation the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. We refer to the number of rows and columns per slide as the stride.

In the current example, the stride is 1, both in terms of height and width. We can also use a larger stride. The figure below shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. We can see that when the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add padding).

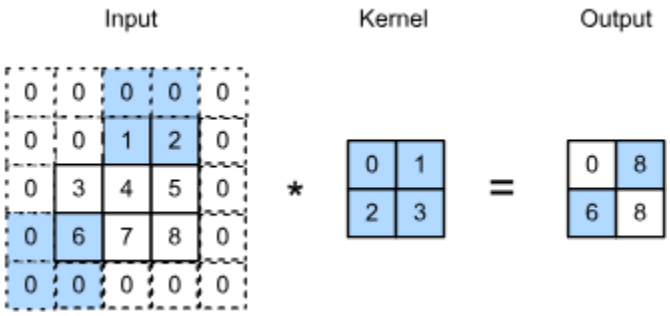


Fig.2.16 Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: 0×0+0×1+1×2+2×3=8,0×0+6×1+0×2+0×3=6

In general, when the stride for the height is sh and the stride for the width is sw, the output shape is

$$\lfloor (nh-kh+ph+sh)/sh \rfloor \times \lfloor (nw-kw+pw+sw)/sw \rfloor.$$

2.8.4 Pooling

As we process images we will eventually want to reduce the resolution of the images. After all, we typically want to output an estimate that does not depend on the dimensionality of the original image. Secondly, when detecting lower-level features, such as edge detection.

Like convolutions, pooling computes the output for each element in a fixed-shape window (also known as a pooling window) of input data. Different from the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer directly calculates the maximum or average value of the elements in the pooling window. These operations are called maximum pooling or average pooling respectively. In maximum pooling, the pooling window starts from the top left of the input array, and slides in the input array from left to right and top to bottom. When the pooling window slides to a certain position, the maximum value of the input subarray in the window is the element at the corresponding location in the output array.

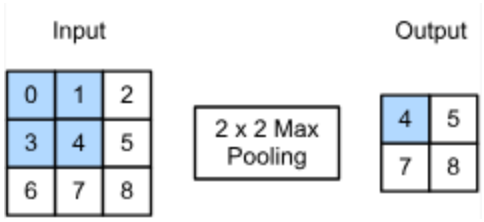


Fig.2.17 Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: max(0,1,3,4)=4

The output array in the figure above has a height of 2 and a width of 2. The four elements are derived from the maximum value of **max**:

$$\max(0,1,3,4)=4,$$

$$\max(1,2,4,5)=5,$$

$$\max(3,4,6,7)=7,$$

$$\max(4,5,7,8)=8.$$

Average pooling works like maximum pooling, only with the maximum operator replaced by the average operator. The pooling layer with a pooling window shape of p×q is called the p×q pooling layer. The pooling operation is called p×q pooling.

2.9 POPULAR CNN ARCHITECTURES

2.9.1 LeNet

LeNet is divided into two parts: a block of convolutional layers and one of fully connected ones.

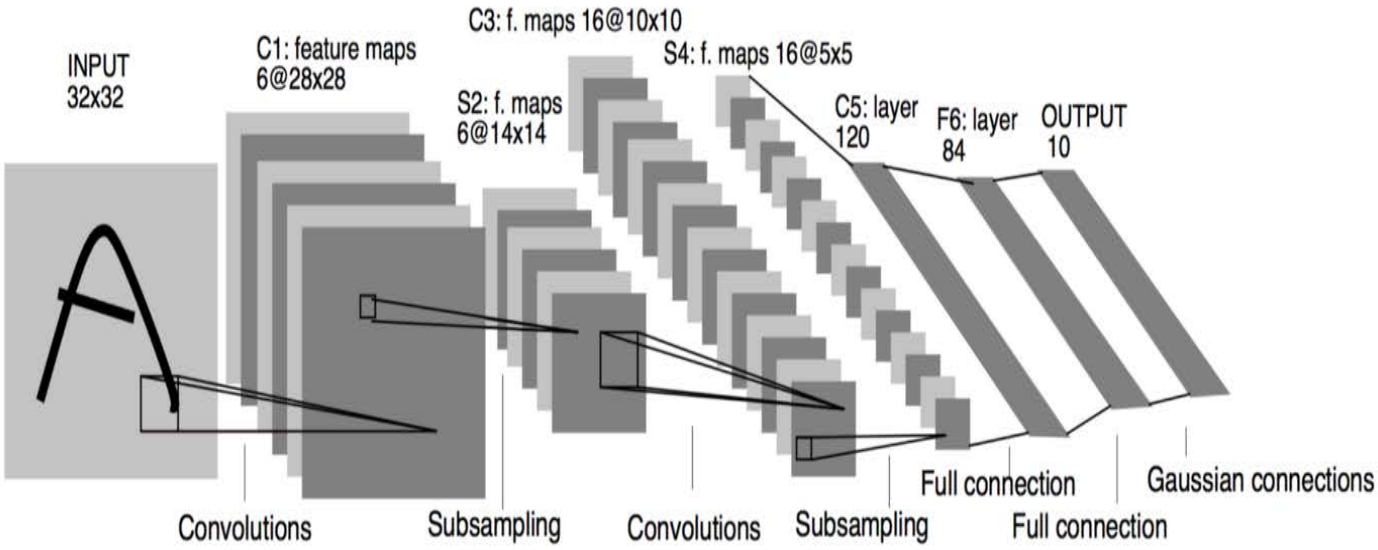


Fig.2.18 LeNet-5

The basic units in the convolutional block are a convolutional layer and a subsequent average pooling layer (note that max-pooling works better, but it had not been invented in the 90s yet). The convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects, and the subsequent average pooling layer is used to reduce the dimensionality. The convolutional layer block is composed of repeated stacks of these two basic units. In the convolutional layer block, each convolutional layer uses a 5×5window and a sigmoid activation function for the output (note that ReLU works better, but it had not been invented in the 90s yet). The number of output channels for the first convolutional layer is 6, and the number of output channels for the second convolutional layer is increased to 16. This is because the height and width of the input of the second convolutional layer is smaller than that of the first convolutional layer. Therefore, increasing the number of output channels makes the parameter sizes of the two convolutional layers similar. The window shape for the two average pooling layers of the convolutional layer block is 2×2 and the stride is 2. Because the pooling window has the same shape as

the stride, the areas covered by the pooling window sliding on each input do not overlap. In other words, the pooling layer performs down sampling.

The output shape of the convolutional layer block is (batch size, channel, height, width). When the output of the convolutional layer block is passed into the fully connected layer block, the fully connected layer block flattens each example in the mini-batch. That is to say, the input shape of the fully connected layer will become two dimensional: the first dimension is the example in the mini-batch, the second dimension is the vector representation after each example is flattened, and the vector length is the product of channel, height, and width. The fully connected layer block has three fully connected layers. They have 120, 84, and 10 outputs, respectively. Here, 10 is the number of output classes.

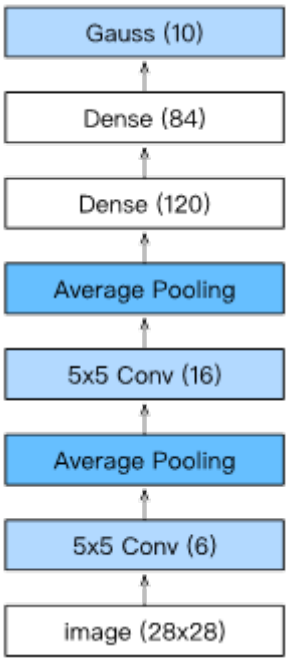


Fig.2.19 Compressed notation for LeNet5

2.9.2 AlexNet

AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the eponymous paper. AlexNet uses an 8-layer convolutional neural network and won the ImageNet Large Scale Visual

Recognition Challenge 2012 with a large margin. This network proved, for the first time, that the features obtained by learning can transcend manually-design features, breaking the previous paradigm in computer vision. The architectures of AlexNet and LeNet are *very similar*, as the diagram below illustrates. Note that we provide a slightly streamlined version of AlexNet which removes the quirks that were needed in 2012 to make the model fit on two small GPUs.

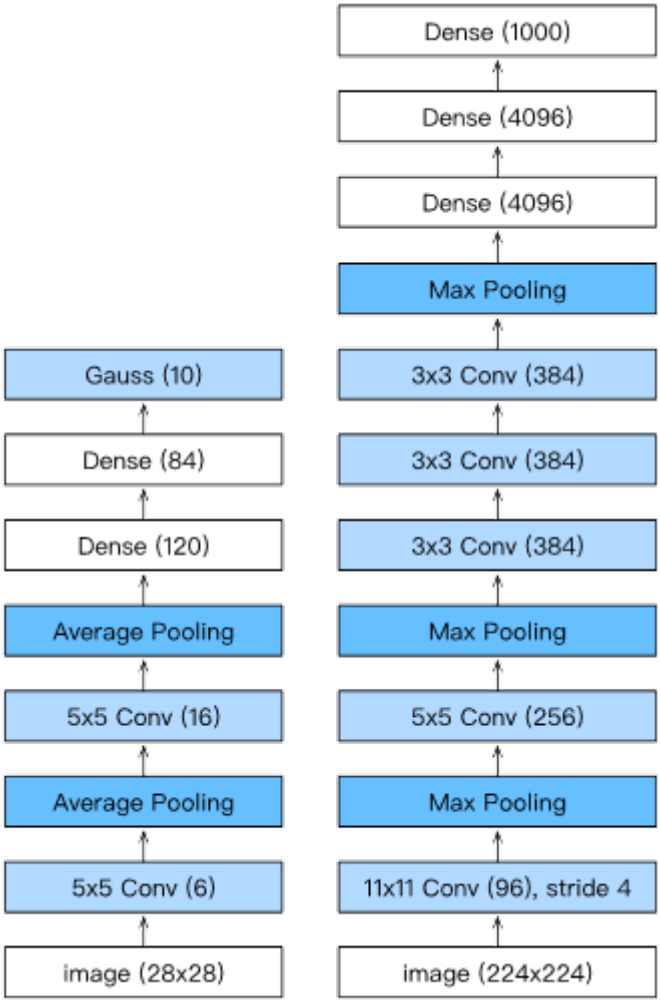


Fig.2.20 LeNet (left) vs AlexNet (right)

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers, five convolutional layers, two fully connected hidden layers, and one fully connected output layer. Second, AlexNet used the ReLU instead of the sigmoid as its activation function. This improved convergence during training significantly.

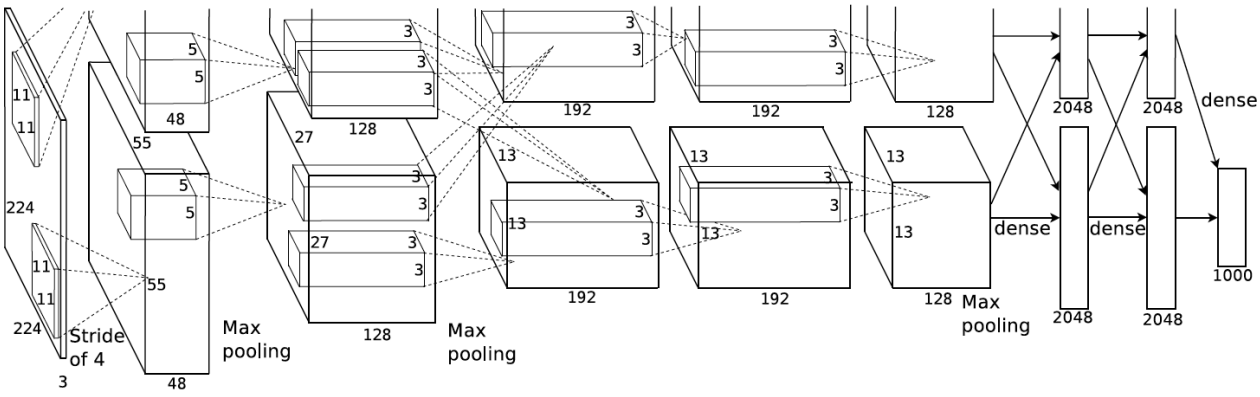


Fig.2.21 AlexNet

In AlexNet's first layer, the convolution window shape is 11×1111×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet images take up more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×55×5, followed by 3×33×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×33×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet. After the last convolutional layer are two fully connected layers with 4096 outputs. These two huge fully connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that one GPU only needs to process half of the model. Fortunately, GPU memory has developed tremendously over the past few years, so we usually do not need this special design anymore (our model deviates from the original paper in this aspect).

Second, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On the one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that back propagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always

1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

2.9.3 VGG – 16

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in **ImageNet**, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to **ILSCRC-2014.** It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. VGG16 was trained for weeks and was using NVIDIA Titan Black GPU’s.

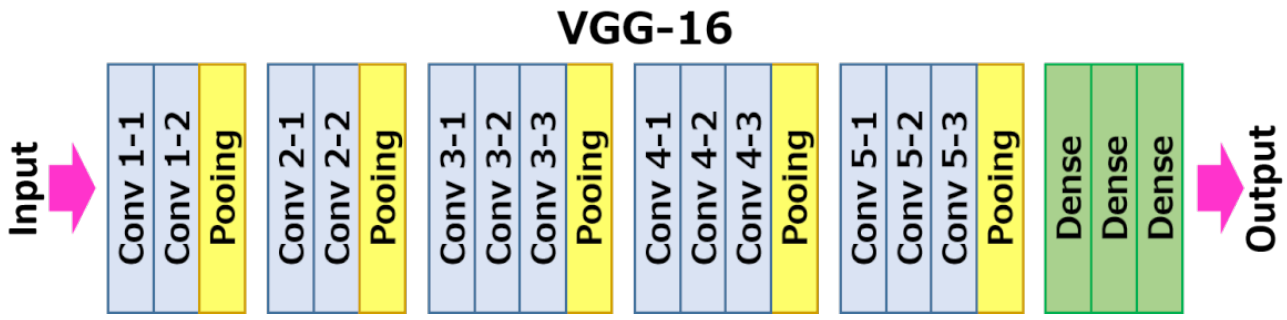


Fig.2.22 VGG-16 Block Diagram

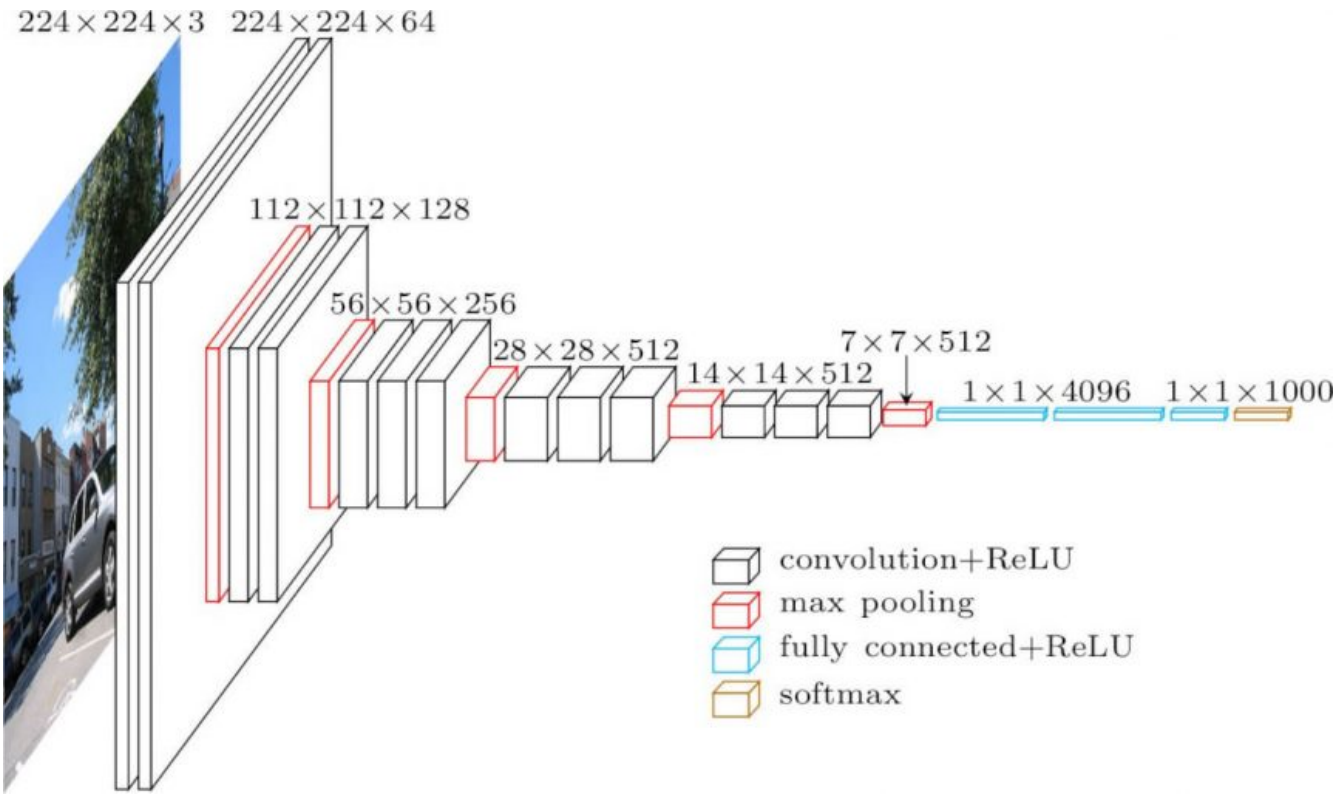


Fig.2.23 VGG-16 Architecture

The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3 (which is the smallest size to capture the notion of left/right, up/down, center). In one of the configurations, it also utilizes 1×1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity). The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution, i.e. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers (not all the conv. layers are followed by max-pooling). Max-pooling is performed over a 2×2 pixel window, with stride 2.

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

All hidden layers are equipped with the rectification (ReLU) non-linearity. It is also noted that none of the networks (except for one) contain Local Response Normalisation (LRN), such normalization does not improve the performance on the ILSVRC dataset, but leads to increased memory consumption and computation time.

The ConvNet configurations are outlined in figure 02. The nets are referred to their names (A-E). All configurations follow the generic design present in architecture and differ only in the depth: from 11 weight layers in the network A (8 conv. and 3 FC layers) to 19 weight layers in the network E (16 conv. and 3 FC layers). The width of conv. layers (the number of channels) is rather small, starting from 64 in the first layer and then increasing by a factor of 2 after each max-pooling layer, until it reaches 512.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Fig.2.24 VGG – 16 Configuration

Unfortunately, there are two major drawbacks with VGGNet:

1.It is *painfully slow* to train.

2.The network architecture weights themselves are quite large (concerning disk/bandwidth).

Due to its depth and number of fully-connected nodes, VGG16 is over 533MB. This makes deploying VGG a tiresome task.VGG16 is used in many deep learning image classification problems; however, smaller network architectures are often more desirable (such as SqueezeNet, GoogLeNet, etc.). But it is a great building block for learning purpose as it is easy to implement.

VGG16 significantly outperforms the previous generation of models in the ILSVRC-2012 and ILSVRC-2013 competitions. The VGG16 result is also competing for the classification task winner (GoogLeNet with 6.7% error) and substantially outperforms the ILSVRC-2013 winning submission Clarifai, which achieved 11.2% with external training data and 11.7% without it. Concerning the single-net performance, VGG16 architecture achieves the best result (7.0% test error), outperforming a single GoogLeNet by 0.9%.

Method	top-1 val. error (%)	top-5 val. error (%)	top-5 test error (%)
VGG (2 nets, multi-crop & dense eval.)	**23.7**	**6.8**	**6.8**
VGG (1 net, multi-crop & dense eval.)	24.4	7.1	7.0
VGG (ILSVRC submission, 7 nets, dense eval.)	24.7	7.5	7.3
GoogLeNet (Szegedy et al., 2014) (1 net)	-	7.9	
GoogLeNet (Szegedy et al., 2014) (7 nets)	-	**6.7**	
MSRA (He et al., 2014) (11 nets)	-	-	8.1
MSRA (He et al., 2014) (1 net)	27.9	9.1	9.1
Clarifai (Russakovsky et al., 2014) (multiple nets)	-	-	11.7
Clarifai (Russakovsky et al., 2014) (1 net)	-	-	12.5
Zeiler & Fergus (Zeiler & Fergus, 2013) (6 nets)	36.0	14.7	14.8
Zeiler & Fergus (Zeiler & Fergus, 2013) (1 net)	37.5	16.0	16.1
OverFeat (Sermanet et al., 2014) (7 nets)	34.0	13.2	13.6
OverFeat (Sermanet et al., 2014) (1 net)	35.7	14.2	-
Krizhevsky et al. (Krizhevsky et al., 2012) (5 nets)	38.1	16.4	16.4
Krizhevsky et al. (Krizhevsky et al., 2012) (1 net)	40.7	18.2	-

Fig.2.25 Performances of Different State-of-Art models on ImageNet

It was demonstrated that the representation depth is beneficial for the classification accuracy, and that state-of-the-art performance on the ImageNet challenge dataset can be achieved using a conventional ConvNet architecture with substantially increased depth.

2.10 GRAPH THEORY

In computer science, a **graph** is an abstract data type that is meant to implement the undirected graph and directed graph concepts from mathematics; specifically, the field of graph theory.

A graph data structure consists of a finite (and possibly mutable) set of *vertices* (also called *nodes* or *points*), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as *edges* (also called *links* or *lines*), and for a directed graph are also known as *arrows*. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some *edge value*, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.)

In one restricted but very common sense of the term, a *graph* is an ordered pair $G = (V, E)$ comprising

- V a set of *vertices* (also called *nodes* or *points*);
- $E \subseteq \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$ a set of *edges* (also called *links* or *lines*), which are unordered pairs of vertices (i.e., an edge is associated with two distinct vertices).

To avoid ambiguity, this type of object may be called precisely an *undirected simple graph*.

In the edge $\{x, y\}$, the vertices x and y are called the *ends* or *end vertices* of the edge. The edge is said to *join* x and y and to be *incident* on x and on y. A vertex may exist in a graph and not belong to an edge. A *loop* is an edge that joins a vertex to itself. *Multiple edges* are two or more edges that join the same two vertices.

In one more general sense of the term allowing multiple edges a *graph* is an ordered triple $G = (V, E, \phi)$ comprising

- V a set of *vertices* (also called *nodes* or *points*);
- E a set of *edges* (also called *links* or *lines*);

- $\phi: E \to \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$ an *incidence function* mapping every edge to an unordered pair of vertices (i.e., an edge is associated with two distinct vertices).

To avoid ambiguity, this type of object may be called precisely an *undirected multigraph*.

Graphs as defined in the two definitions above cannot have loops, because a loop joining a vertex x is the edge (for an undirected simple graph) or is incident on (for an undirected multigraph) $\{x, x\} = \{x\}$ which is not in $\{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$. So to allow loops the definitions must be expanded. For undirected simple graphs, $E \subseteq \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$ should become $E \subseteq \{\{x, y\} \mid (x, y) \in V^2\}$. For undirected multigraphs, $\phi: E \to \{\{x, y\} \mid (x, y) \in V^2 \wedge x \neq y\}$ should become $\phi: E \to \{\{x, y\} \mid (x, y) \in V^2\}$. To avoid ambiguity, these types of objects may be called precisely an *undirected simple graph with loops* and an *undirected multigraph with loops* respectively.

V and E are usually taken to be finite, and many of the well-known results are not true (or are rather different) for *infinite graphs* because many of the arguments fail in the infinite case. Moreover, V is often assumed to be non-empty, but E is allowed to be the empty set. The *order* of a graph is $|V|$, its number of vertices. The *size* of a graph is $|E|$, its number of edges. The *degree* or *valency* of a vertex is the number of edges that are incident to it, where a loop is counted twice.

In an undirected simple graph of order n, the maximum degree of each vertex is $n - 1$ and the maximum size of the graph is $n(n - 1)/2$.

The edges E of an undirected graph G induce a symmetric binary relation $\sim$ on V that is called the *adjacency relation* of G. Specifically, for each edge $\{x, y\}$, its end vertices x and y are said to be *adjacent* to one another, which is denoted $x \sim y$.

2.10.1 Directed Graph

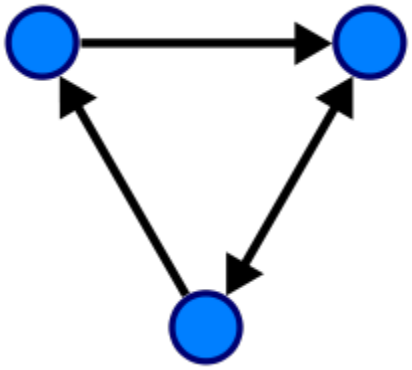


Fig.2.26 A Directed graph with 3 vertices and 3 edges

A *directed graph* or *digraph* is a graph in which edges have orientations.

In one restricted but very common sense of the term, a *directed graph* is an ordered pair $G = (V, A)$ (sometimes $G = (V, E)$) comprising

- V a set of *vertices* (also called *nodes* or *points*);
- $A \subseteq \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$ a set of *edges* (also called *directed edges*—sometimes simply *edges* with the corresponding set named E instead of A—>, *directed links* or *directed lines*), which are ordered pairs of *distinct* vertices (i.e., an edge is associated with two distinct vertices).

To avoid ambiguity, this type of object may be called precisely a *directed simple graph*.

In the edge (x, y) directed *from x to y*, x is called the *tail* of the edge and y the *head* of the edge. The edge (y, x) is called the *inverted edge* of (x, y).

In one more general sense of the term allowing multiple edges,[9] a *directed graph* is an ordered triple $G = (V, A, \phi)$ (sometimes $G = (V, E, \phi)$) comprising

- V a set of *vertices* (also called *nodes* or *points*);
- A a set of *edges* (also called *directed edges*—sometimes simply *edges* with the corresponding set named E instead of A—, *directed links* or *directed lines*);

- $\phi: A \rightarrow \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$ an *incidence function* mapping every edge to an ordered pair of *distinct* vertices (i.e., an edge is associated with two distinct vertices).

To avoid ambiguity, this type of object may be called precisely a *directed multigraph*.

Directed graphs as defined in the two definitions above cannot have loops, because a loop joining a vertex x is the edge (for a directed simple graph) or is incident on (for a directed multigraph) (x, x) which is not in $\{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$. So to allow loops the definitions must be expanded. For directed simple graphs, $A \subseteq \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$ should become $A \subseteq V^2$. For directed multigraphs, $\phi: A \rightarrow \{(x, y) \mid (x, y) \in V^2 \wedge x \neq y\}$ should become $\phi: A \rightarrow V^2$. To avoid ambiguity, these types of objects may be called precisely a *directed simple graph with loops* and a *directed multigraph with loops* (or a *quiver*) respectively.

A directed simple graph with loops is a homogeneous relation (a binary relation between a set and itself). A directed simple graph with loops $G = (V, A)$ is said *symmetric* if, for every edge in A, the corresponding inverted edge also belongs to A.

2.10.2 Mixed Graph

A *mixed graph* is a graph in which some edges may be directed and some may be undirected. It is an ordered triple $G = (V, E, A)$ for a *mixed simple graph* and $G = (V, E, A, \phi_E, \phi_A)$for a *mixed multigraph* with V, E, A, ϕ_E and ϕ_A defined as above. Directed and undirected graphs are special cases.

2.10.3 Weighted Graph

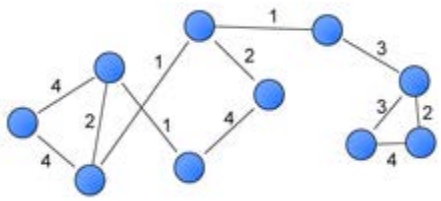


Fig.2.27 A weighted graph with 10 vertices and 12 edges

A weighted graph or a *network* is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand. Such graphs arise in many contexts, for example in shortest path problems such as the traveling salesman problem.

2.11 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm (or **Dijkstra's Shortest Path First algorithm**, **SPF algorithm**) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and edge path costs represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithm is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF). It is also employed as a subroutine in other algorithms such as Johnson's.

The Dijkstra algorithm uses labels that are positive integer or real numbers, which have the strict weak ordering defined. Interestingly, Dijkstra can be generalized to use labels defined in any way, provided they have the strict partial order defined, and provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the Generic Dijkstra shortest-path algorithm.

Dijkstra's original algorithm does not use a min-priority queue and runs in time (where n is the number of nodes). The idea of this algorithm is also given in Leyzorek et al. 1957. The implementation based on a min-priority queue implemented by a Fibonacci heap and running in E(where E is the number of edges) is due to Fredman & Tarjan 1984. This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

However, specialized cases (such as bounded/integer weights, directed acyclic graphs etc.) can indeed be improved further as detailed in § Specialized variants.

In some fields, artificial intelligence in particular, Dijkstra's algorithm or a variant of it is known as **uniform cost search** and formulated as an instance of the more general idea of best-first search.

2.11.1 Algorithm

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.

2. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Set the initial node as current.[13]

3. For the current node, consider all of its unvisited neighbours and calculate their *tentative* distances through the current node. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbour *B* has length 2, then the distance to *B* through *A* will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.

4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

When planning a route, it is actually not necessary to wait until the destination node is "visited" as above: the algorithm can stop once the destination node has the smallest tentative distance among all "unvisited" nodes (and thus could be selected as the next "current").

2.11.2 Description

Suppose you would like to find the *shortest path* between two intersections on a city map: a *starting point* and a *destination*. Dijkstra's algorithm initially marks the distance (from the starting point) to every other intersection on the map with *infinity*. This is done not to imply that there is an infinite distance, but to note that those intersections have not been visited yet. Some variants of this method leave the intersections' distances *unlabelled*. Now select the *current intersection* at each iteration. For the first iteration, the current intersection will be the starting point, and the distance to it (the intersection's label) will be *zero*. For subsequent iterations (after the first), the current intersection will be a *closest unvisited intersection* to the starting point (this will be easy to find).

From the current intersection, *update* the distance to every unvisited intersection that is directly connected to it. This is done by determining the *sum* of the distance between an unvisited intersection and the value of the current intersection and then relabelling the unvisited intersection with this value (the sum) if it is less than the unvisited intersection's current value. In effect, the intersection is relabelled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabelled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighbouring intersection, mark the current intersection as *visited* and select an unvisited intersection with minimal distance (from the starting point) – or the lowest label—as the current intersection. Intersections marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighbouring intersections with the shortest distances, marking the current intersection as visited, and moving onto a closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection), you have determined the shortest path to it from the starting point and can *trace your way back following the arrows in reverse*. In the algorithm's implementations, this is usually done (after the algorithm has reached the destination node) by following the nodes' parents from the destination node up to the starting node; that's why we also keep track of each node's parent.

This algorithm makes no attempt of direct "exploration" towards the destination as one might expect. Rather, the sole consideration in determining the next "current" intersection is its distance from the starting point. This algorithm therefore expands outward from the starting point, interactively considering every node that is closer in terms of shortest path distance until it reaches the destination. When understood in this way, it is clear how the algorithm necessarily finds the shortest path. However, it may also reveal one of the algorithm's weaknesses: its relative slowness in some topologies.

2.11.3 Pseudocode

In the following algorithm, the code u ← vertex in *Q* with min dist[u], searches for the vertex *u* in the vertex set *Q* that has the least dist[*u*] value. length(*u*, *v*) returns the length of the edge joining (i.e. the distance between) the two neighbor-nodes *u* and *v*. The variable *alt* on line 18 is the length of the path from the root node to the neighbor node *v*if it were to go through *u*. If this path is shorter than the current shortest path recorded for *v*, that current path is replaced with this *alt* path. The prev array is populated with a pointer to the "next-hop" node on the source graph to get the shortest route to the source.

```
 1  function Dijkstra(Graph, source):
 2
 3      create vertex set Q
 4
 5      for each vertex v in Graph:
 6          dist[v] ← INFINITY
 7          prev[v] ← UNDEFINED
 8          add v to Q
10      dist[source] ← 0
11
12      while Q is not empty:
13          u ← vertex in Q with min dist[u]
14
15          remove u from Q
16
17          for each neighbor v of u:           // only v that are still in Q
18              alt ← dist[u] + length(u, v)
```

```
19          if alt < dist[v]:
20              dist[v] ← alt
21              prev[v] ← u
22
23      return dist[], prev[]
```

If we are only interested in a shortest path between vertices *source* and *target*, we can terminate the search after line 15 if *u* = *target*. Now we can read the shortest path from *source* to *target* by reverse iteration:

```
1  S ← empty sequence
2  u ← target
3  if prev[u] is defined or u = source:          // Do something only if the vertex is reachable
4      while u is defined:                       // Construct the shortest path with a stack S
5          insert u at the beginning of S        // Push the vertex onto the stack
6          u ← prev[u]                           // Traverse from target to source
```

Now sequence *S* is the list of vertices constituting one of the shortest paths from *source* to *target*, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between *source* and *target* (there might be several different ones of the same length). Then instead of storing only a single node in each entry of prev[] we would store all nodes satisfying the relaxation condition. For example, if both *r* and *source* connect to *target* and both of them lie on different shortest paths through *target* (because the edge cost is the same in both cases), then we would add both *r* and *source* to prev[*target*]. When the algorithm completes, prev[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the new graph will be the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these shortest paths between two given nodes we would use a path finding algorithm on the new graph, such as **depth-first search**.

2.11.3.a Using Priority Queue

A min-priority queue is an abstract data type that provides 3 basic operations : add_with_priority(), decrease_priority() and extract_min(). As mentioned earlier, using such a data structure can lead to faster computing times than using a basic queue. Notably, Fibonacci heap (Fredman & Tarjan 1984) or Brodal queue offer optimal implementations for those 3 operations. As the algorithm is slightly different, we mention it here, in pseudo-code as well:

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0                  // Initialization
3
4      create vertex set Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY             // Unknown distance from source to v
9          prev[v] ← UNDEFINED                // Predecessor of v
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:                  // The main loop
15         u ← Q.extract_min()                // Remove and return best vertex
16         for each neighbor v of u:          // only v that are still in Q
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

Instead of filling the priority queue with all nodes in the initialization phase, it is also possible to initialize it to contain only *source*; then, inside the **if** *alt* < dist[*v*] block, the node must be inserted if not already in the queue (instead of performing a *decrease_priority* operation).

Other data structures can be used to achieve even faster computing times in practice.

2.11.4 Running Time

Every time the main loop executes, one vertex is extracted from the queue. Assuming that there are V vertices in the graph, the queue may contain O(V) vertices. Each pop operation takes O(log V) time assuming the heap implementation of priority queues. So the total time required to execute the main loop itself is O(V log V). In addition, we must consider the time spent in the function expand, which applies the function handle_edge to each outgoing edge. Because expand is only called once per vertex, handle_edge is only called once per edge. It might call push(v'), but there can be at most V such calls during the entire execution, so the total cost of that case arm is at most O(V log V). The other case arm may be called O(E) times, however, and each call to increase_priority takes O(log V) time with the heap implementation. Therefore the total run time is O(V log V + E log V), which is O(E log V) because V is O(E) assuming a connected graph.

(There is another more complicated priority-queue implementation called a Fibonacci heap that implements increase_priority in O(1) time, so that the asymptotic complexity of Dijkstra's algorithm becomes O(V log V + E); however, large constant factors make Fibonacci heaps impractical for most uses.)

$$O(|E|+|V|\log|V|)$$

3.TECHNICAL SPECIFICATIONS

3.1 HARDWARE SPECIFICATIONS

3.1.1 Raspberry Pi

The **Raspberry Pi** is a series of small single-board computers developed in the United Kingdom by the Raspberry Pi Foundation to promote teaching of basic computer science in schools and in developing countries. The original model became far more popular than anticipated, selling outside its target market for uses such as robotics. We chose Raspberry Pi 3 Model B because of the Linux based OS – Raspbian and programmable GPIOs. **Raspberry Pi 3 Model B** has a 1.2 GHz 64-bit quad core processor, on-board WiFi, Bluetooth and USB boot capabilities.

Fig.3.1 Raspberry Pi3 Model B

3.1.1.a Hardware specifications:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port

- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Upgraded switched Micro USB power source up to 2.5A

The OS runs from the SD card. NOOBS is an easy operating system installer which contains Raspbian and LibreELEC. It also provides a selection of alternative operating systems which are then downloaded from the internet and installed. Once the OS is setup, the board is ready to use.

3.1.1.b GPIOs

A powerful feature of the Raspberry Pi is the row of GPIO (general-purpose input/output) pins along the top edge of the board. A 40-pin GPIO header is found on all current Raspberry Pi boards.

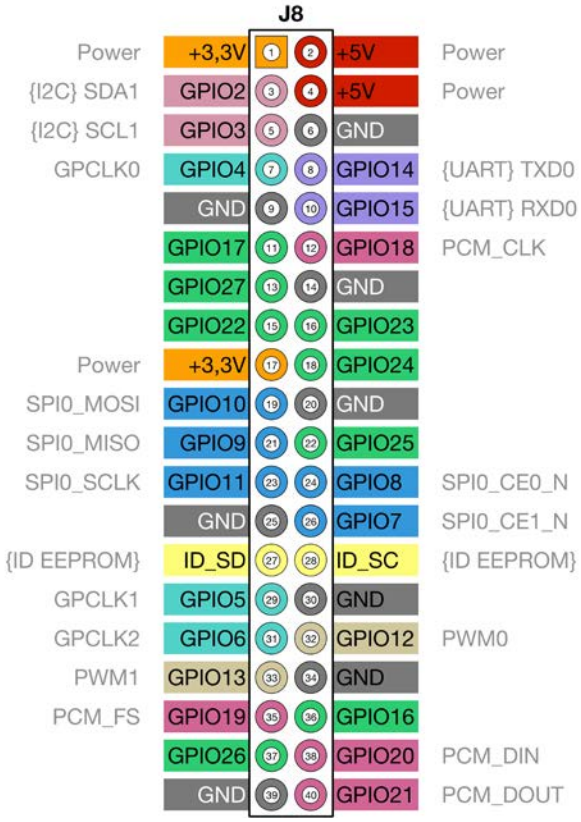


Fig.3.2 GPIO Pinout of Raspberry Pi3 Model B

3.1.1.c Voltages

Two 5V pins and two 3V3 pins are present on the board, as well as a number of ground pins (0V), which are unconfigurable. The remaining pins are all general purpose 3V3 pins, meaning outputs are set to 3V3 and inputs are 3V3-tolerant.

3.1.1.d Outputs

A GPIO pin designated as an output pin can be set to high (3V3) or low (0V).

3.1.1.e Inputs

A GPIO pin designated as an input pin can be read as high (3V3) or low (0V). This is made easier with the use of internal pull-up or pull-down resistors. Pins GPIO2 and GPIO3 have fixed pull-up resistors, but for other pins this can be configured in software.

As well as simple input and output devices, the GPIO pins can be used with a variety of alternative functions, some are available on all pins, others on specific pins.

- PWM (pulse-width modulation)
    - Software PWM available on all pins
    - Hardware PWM available on GPIO12, GPIO13, GPIO18, GPIO19
- SPI
    - SPI0: MOSI (GPIO10); MISO (GPIO9); SCLK (GPIO11); CE0 (GPIO8), CE1 (GPIO7)
    - SPI1: MOSI (GPIO20); MISO (GPIO19); SCLK (GPIO21); CE0 (GPIO18); CE1 (GPIO17); CE2 (GPIO16)
- I2C
    - Data: (GPIO2); Clock (GPIO3)
    - EEPROM Data: (GPIO0); EEPROM Clock (GPIO1)
- Serial
    - TX (GPIO14); RX (GPIO15)

It is possible to control GPIO pins using a number of programming languages and tools, including Python using the RPi.GPIO package.

3.1.2 Raspberry Pi Camera v1

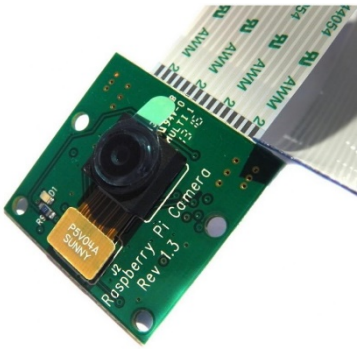


Fig.3.3 Raspberry Pi Camera v1

The Raspberry Pi Camera Module v1 is a official product from the Raspberry Pi Foundation. This 5-megapixel model was released in 2013 that has a 5-megapixel OmniVision OV5647 sensor. The Camera Module can be used to take high-definition video, as well as stills photographs. It supports 1080p30, 720p60 and VGA90 video modes, as well as still capture. It attaches via a 15cm ribbon cable to the CSI port on the Raspberry Pi. The camera works with all models of Raspberry Pi 1, 2, and 3. It can be accessed through the MMAL and V4L APIs, and there are numerous third-party libraries built for it, including the Picamera Python library.

3.1.3 MG995

This high-speed metal gear dual ball bearing servo motor unit comes with 30cm wire with 3 pin ‘S’ type female header connector that fits most receivers. This high-speed standard servo can rotate approximately 120 degrees (60 in each direction). You can use any servo code, hardware or library to control these servos.

Fig.3.4 MG995 Servo Motor

Specifications:

- Weight: 55 g
- Dimension: 40.7 x 19.7 x 42.9 mm approx.
- Stall torque: 8.5 kgf·cm (4.8 V), 10 kgf·cm (6 V)
- Operating speed: 0.2 s/60º (4.8 V), 0.16 s/60º (6 V)
- Operating voltage: 4.8 V a 7.2 V
- Dead band width: 5 µs
- Stable and shock proof double ball bearing design
- Temperature range: 0 ºC – 55 ºC

3.1.4 DC Motor

Fig.3.5 DC Motor

The 100 RPM Dual Shaft BO Motor Plastic Gear Motor – BO series straight motor gives good torque and rpm at lower operating voltages, which is the biggest advantage of these motors. Small shaft with

matching wheels gives an optimized design for your application or robot. Mounting holes on the body & light weight makes it suitable for in-circuit placement. Low-cost geared DC Motor. It is an alternative to metal gear DC motors. It comes with an operating voltage of 3-12V and is perfect for building small and medium robots. The motor is ideal for DIY enthusiasts. This motor set is inexpensive, small, easy to install, and ideally suited for use in a mobile robot car.

Specifications:

- Operating Voltage (DC): 3~12V
- Shaft Length (mm): 8.5
- Shaft Diameter (mm): 5.5
- No Load Current: 40-180mA
- Rated Torque: 0.35 Kg-cm
- Weight (gm): 30

3.1.5 L298N Driver

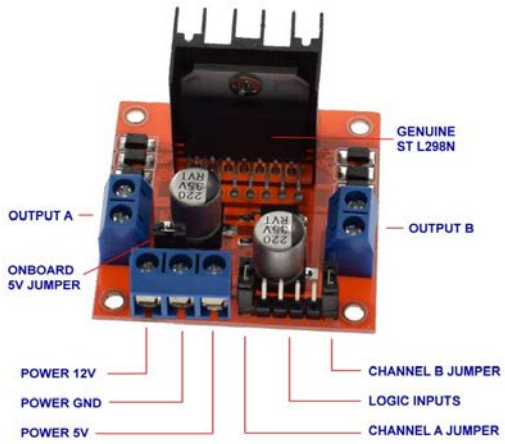


Fig.3.6 L298N Driver

The L298N is a dual H-Bridge motor driver which allows speed and direction control of two DC motors at the same time. The module can drive DC motors that have voltages between 5 and 35V, with a peak current up to 2A.

Let’s take a closer look at the pinout of L298N module and explain how it works. The module has two screw terminal blocks for the motor A and B, and another screw terminal block for the Ground pin, the VCC for motor and a 5V pin which can either be an input or output.
This depends on the voltage used at the motors VCC. The module have an onboard 5V regulator which is either enabled or disabled using a jumper. If the motor supply voltage is up to 12V we can enable the 5V regulator and the 5V pin can be used as output, for example for powering our Arduino board. But if the motor voltage is greater than 12V we must disconnect the jumper because those voltages will cause damage to the onboard 5V regulator. In this case the 5V pin will be used as input as we need connect it to a 5V power supply in order the IC to work properly.

We can note here that this IC makes a voltage drop of about 2V. So for example, if we use a 12V power supply, the voltage at motors terminals will be about 10V, which means that we won’t be able to get the maximum speed out of our 12V DC motor.

Next are the logic control inputs. The Enable A and Enable B pins are used for enabling and controlling the speed of the motor. If a jumper is present on this pin, the motor will be enabled and work at maximum speed, and if we remove the jumper we can connect a PWM input to this pin and in that way control the speed of the motor. If we connect this pin to a Ground the motor will be disabled.

Next, the Input 1 and Input 2 pins are used for controlling the rotation direction of the motor A, and the inputs 3 and 4 for the motor B. Using these pins we actually control the switches of the H-Bridge inside the L298N IC. If input 1 is LOW and input 2 is HIGH the motor will move forward, and vice versa, if input 1 is HIGH and input 2 is LOW the motor will move backward. In case both inputs are same, either LOW or HIGH the motor will stop. The same applies for the inputs 3 and 4 and the motor B.

On the other hand, for controlling the rotation direction, we just need to inverse the direction of the current flow through the motor, and the most common method of doing that is by using an H-Bridge. An H-Bridge circuit contains four switching elements, transistors or MOSFETs, with the motor at the centre forming an H-like configuration. By activating two particular switches at the same time we can change the direction of the current flow, thus change the rotation direction of the motor.

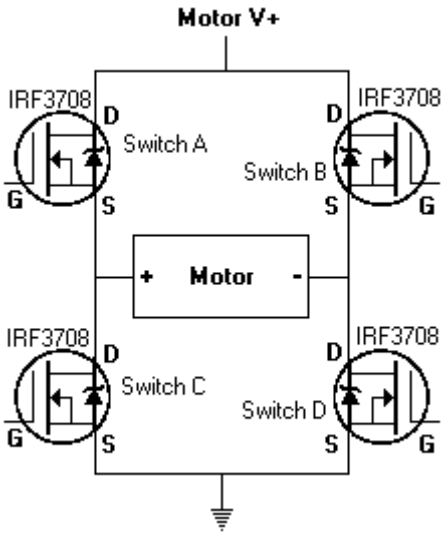


Fig.3.7 H-bridge circuit connected to the motor

So, if we combine these two methods, the PWM and the H-Bridge, we can have a complete control over the DC motor. There are many DC motor drivers that have these features and the L298N is one of them.

3.2 SOFTWARE SPECIFICATIONS

3.2.1 **Raspbian**

Fig.3.8 Raspbian OS

Raspbian is a Debian-based computer operating system for Raspberry Pi. There are several versions of Raspbian including Raspbian Stretch and Raspbian Jessie. Since 2015 it has been officially provided by the Raspberry Pi Foundation as the primary operating system for the family of Raspberry Pi single-board

computers. Raspbian was created by Mike Thompson and Peter Green as an independent project. The initial build was completed in June 2012. The operating system is still under active development. Raspbian is highly optimized for the Raspberry Pi line's low-performance ARM CPUs.

Raspbian uses PIXEL, **P**i **I**mproved **X**-Window **E**nvironment, **L**ightweight as its main desktop environment as of the latest update. It is composed of a modified LXDE desktop environment and the Openbox stacking window manager with a new theme and few other changes.

3.2.2 Python

Fig.3.9 Python logo

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales. Van Rossum led the language community until July 2018.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python features a comprehensive standard library, and is referred to as "batteries included". It also features dynamic name resolution (late binding), which binds method and variable names during program execution. Python interpreters are available for many operating systems. CPython, the reference implementation of Python, is open-source software and has a community-based development model.

For this project, we are using Python 3.6.5 as it supports TensorFlow 1.13 which is used to build the neural network model. Python 3.6 was released on 23 December 2016 and Python 3.6.5 is the fifth maintenance release of Python 3.6.

3.2.3 TensorFlow

Fig.3.10 TensorFlow

TensorFlow is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning (aka neural networking) models and algorithms and makes them useful by way of a common metaphor. It uses Python to provide a convenient front-end API for building applications with the framework, while executing those applications in high-performance C++. Python just directs traffic between the pieces, and provides high-level programming abstractions to hook them together. It is used for both research and production at Google. TensorFlow was developed by the **Google Brain** team for internal Google use. It was released under the Apache 2.0 open-source license on November 9, 2015.

TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on February 11, 2017. While the reference implementation runs on single devices, TensorFlow can run on multiple CPUs and GPUs (with optional **CUDA** and **SYCL** extensions for general-purpose computing on graphics processing units). TensorFlow is available on 64-bit Linux, macOS, Windows, and mobile computing platforms including Android and iOS. Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as *tensors*. During the Google I/O Conference in June 2016, Jeff Dean stated that 1,500 repositories on GitHub mentioned TensorFlow, of which only 5 were from Google.

The single biggest benefit TensorFlow provides for machine learning development is abstraction. Instead of dealing with the nitty-gritty details of implementing algorithms, or figuring out proper ways to hitch the output of one function to the input of another, the developer can focus on the overall logic of the application. TensorFlow takes care of the details behind the scenes.

In November, TensorFlow celebrated its 3rd birthday and TensorFlow 2.0 was announced. TensorFlow 2.0 will focus on simplicity and ease of use, featuring updates like:

- Easy model building with Keras and eager execution.
- Robust model deployment in production on any platform.
- Powerful experimentation for research.
- Simplifying the API by cleaning up deprecated APIs and reducing duplication.

Fig.3.11 TensorFlow 2.0

Keras, a user-friendly API standard for machine learning, has been made the central high-level API used to build and train models. The Keras API makes it easy to get started with TensorFlow. Importantly, Keras provides several model-building APIs (Sequential, Functional, and Subclassing), so you can choose the right level of abstraction for your project. TensorFlow's implementation contains enhancements including eager execution, for immediate iteration and intuitive debugging, and tf.data, for building scalable input pipelines. These new features have been added to versions >1.13.

Here's an example workflow:

- **Load your data using tf.data**. Training data is read using input pipelines which are created using tf.data. Feature characteristics, for example bucketing and feature crosses are described using tf.feature_column. Convenient input from in-memory data (for example, NumPy) is also supported.
- **Build, train and validate your model with tf.keras, or use Premade Estimators**. Keras integrates tightly with the rest of TensorFlow so you can access TensorFlow's features whenever you want. A set of standard packaged models (for example, linear or logistic regression, gradient boosted trees, random forests) are also available to use directly (implemented using the tf.estimator API).

- **Run and debug with eager execution, then use tf.function for the benefits of graphs**. TensorFlow 2.0 runs with eager execution by default for ease of use and smooth debugging. Additionally, the tf.function annotation transparently translates your Python programs into TensorFlow graphs. This process retains all the advantages of 1.x TensorFlow graph-based execution: Performance optimizations, remote execution and the ability to serialize, export and deploy easily, while adding the flexibility and ease of use of expressing programs in simple Python.
- **Use Distribution Strategies for distributed training**. For large ML training tasks, the Distribution Strategy API makes it easy to distribute and train models on different hardware configurations without changing the model definition. Since TensorFlow provides support for a range of hardware accelerators like CPUs, GPUs, and TPUs, you can enable training workloads to be distributed to single-node/multi-accelerator as well as multi-node/multi-accelerator configurations, including TPU Pods. Although this API supports a variety of cluster configurations, templates to deploy training on Kubernetes clusters in on-prem or cloud environments are provided.
- **Export to SavedModel**. TensorFlow will standardize on SavedModel as an interchange format for TensorFlow Serving, TensorFlow Lite, TensorFlow.js, TensorFlow Hub, and more.

3.2.4 OpenCV

Fig.3.12 OpenCV

OpenCV (Open source computer vision) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then Itseez (which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license. Officially launched in 1999, the OpenCV project was initially an **Intel Research** initiative to advance CPU-intensive applications, part of a series of projects including real-time ray tracing and 3D display walls. The main contributors to the project included a number of

optimization experts in Intel Russia, as well as Intel's Performance Library Team. In the early days of OpenCV, the goals of the project were described as:

- Advance vision research by providing not only open but also optimized code for basic vision infrastructure. No more reinventing the wheel.
- Disseminate vision knowledge by providing a common infrastructure that developers could build on, so that code would be more readily readable and transferable.
- Advance vision-based commercial applications by making portable, performance-optimized code available for free – with a license that did not require code to be open or free itself.

The first alpha version of OpenCV was released to the public at the IEEE Conference on Computer Vision and Pattern Recognition in 2000, and five betas were released between 2001 and 2005. The first 1.0 version was released in 2006. A version 1.1 "pre-release" was released in October 2008.

The second major release of the OpenCV was in October 2009. OpenCV 2 includes major changes to the C++ interface, aiming at easier, more type-safe patterns, new functions, and better implementations for existing ones in terms of performance (especially on multi-core systems). Official releases now occur every six months and development is now done by an independent Russian team supported by commercial corporations.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 18 million. The library is used extensively in companies, research groups and by governmental bodies.

Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many start-ups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV. OpenCV’s deployed uses span the range from stitching streetview images together, detecting intrusions in surveillance video in Israel, monitoring mine equipment in China, helping robots navigate and pick up objects at Willow Garage, detection of swimming pool drowning accidents in Europe, running interactive art in Spain and New York, checking runways for debris in Turkey, inspecting labels on products in factories around the world on to rapid face detection in Japan.

It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available. A full-featured CUDA and OpenCL interfaces are being actively developed right now. There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers.

4. SIMULATION

In a new automotive application, NVIDIA have used convolutional neural networks (CNNs) to map the raw pixels from a front-facing camera to the steering commands for a self-driving car. This powerful end-to-end approach means that with minimum training data from humans, the system learns to steer, with or without lane markings, on both local roads and highways. The system can also operate in areas with unclear visual guidance such as parking lots or unpaved roads.

NVIDIA designed the end-to-end learning system using an NVIDIA DevBox running Torch 7 for training. An NVIDIA DRIVETM PX self-driving car computer, also with Torch 7, was used to determine where to drive—while operating at 30 frames per second (FPS). The system is trained to automatically learn the internal representations of necessary processing steps, such as detecting useful road features, with only the human steering angle as the training signal. NVIDIA never explicitly trained it to detect, for example, the outline of roads. In contrast to methods using explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously.

NVIDIA believe that end-to-end learning leads to better performance and smaller systems. Better performance results because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e. g., lane detection. Such criteria understandably are selected for ease of human interpretation which doesn’t automatically guarantee maximum system performance. Smaller networks are possible because the system learns to solve the problem with the minimal number of processing steps.

4.1 CONVOLUTIONAL NEURAL NETWORKS TO PROCESS VISUAL DATA

CNNs have revolutionized the computational pattern recognition process. Prior to the widespread adoption of CNNs, most pattern recognition tasks were performed using an initial stage of hand-crafted feature extraction followed by a classifier. The important breakthrough of CNNs is that features are now learned automatically from training examples. The CNN approach is especially powerful when applied to image recognition tasks because the convolution operation captures the 2D nature of images. By using the convolution kernels to scan an entire image, relatively few parameters need to be learned compared to the total number of operations.

While CNNs with learned features have been used commercially for over twenty years, their adoption has exploded in recent years because of two important developments. First, large, labeled data sets such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) are now widely available for training and validation. Second, CNN learning algorithms are now implemented on massively parallel graphics processing units (GPUs), tremendously accelerating learning and inference ability.

The CNNs that we describe here go beyond basic pattern recognition. We developed a system that learns the entire processing pipeline needed to steer an automobile. The groundwork for this project was actually done over 10 years ago in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE), in which a sub-scale radio control (RC) car drove through a junk-filled alley way. DAVE was trained on hours of human driving in similar, but not identical, environments. The training data included video from two cameras and the steering commands sent by a human operator.

In many ways, DAVE was inspired by the pioneering work of Pomerleau, who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. ALVINN is a precursor to DAVE, and it provided the initial proof of concept that an end-to-end trained neural network might one day be capable of steering a car on public roads. DAVE demonstrated the potential of end-to-end learning, and indeed was used to justify starting the DARPA Learning Applied to Ground Robots (LAGR) program, but DAVE's performance was not sufficiently reliable to provide a full alternative to the more modular approaches to off-road driving. (DAVE's mean distance between crashes was about 20 meters in complex environments.)

About a year ago we started a new effort to improve on the original DAVE, and create a robust system for driving on public roads. The primary motivation for this work is to avoid the need to recognize specific human-designated features, such as lane markings, guard rails, or other cars, and to avoid having to create a collection of "if, then, else" rules, based on observation of these features. We are excited to share the preliminary results of this new effort, which is aptly named: DAVE–2.

4.2 THE DAVE-2 SYSTEM

Figure 2 shows a simplified block diagram of the collection system for training data of DAVE-2. Three cameras are mounted behind the windshield of the data-acquisition car, and timestamped video from the cameras is captured simultaneously with the steering angle applied by the human driver. The steering command is obtained by tapping into the vehicle's Controller Area Network (CAN) bus. In order to make

our system independent of the car geometry, we represent the steering command as $1/r$, where r is the turning radius in meters. We use $1/r$ instead of r to prevent a singularity when driving straight (the turning radius for driving straight is infinity). $1/r$ smoothly transitions through zero from left turns (negative values) to right turns (positive values).

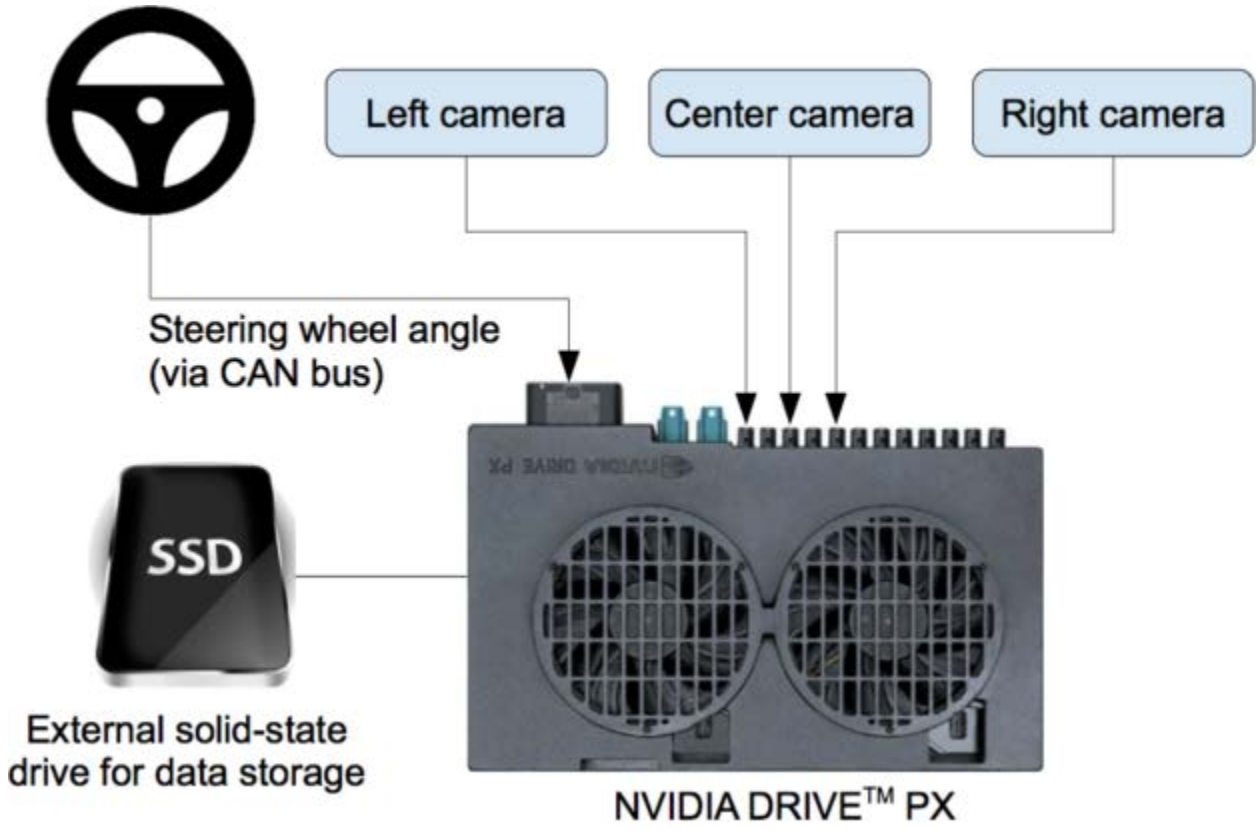


Fig.4.1 High-level view of data collection system

Training data contains single images sampled from the video, paired with the corresponding steering command (1/r). Training with data from only the human driver is not sufficient; the network must also learn how to recover from any mistakes, or the car will slowly drift off the road. The training data is therefore augmented with additional images that show the car in different shifts from the center of the lane and rotations from the direction of the road.

The images for two specific off-center shifts can be obtained from the left and the right cameras.

Additional shifts between the cameras and all rotations are simulated through viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don't have, so we approximate the transformation by assuming all points below the horizon are on flat ground, and all points above the horizon are infinitely far away. This works fine for flat

terrain, but for a more complete rendering it introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately these distortions don’t pose a significant problem for network training. The steering label for the transformed images is quickly adjusted to one that correctly steers the vehicle back to the desired location and orientation in two seconds.

Figure 3 shows a block diagram of our training system. Images are fed into a CNN that then computes a proposed steering command. The proposed command is compared to the desired command for that image, and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation as implemented in the Torch 7 machine learning package.

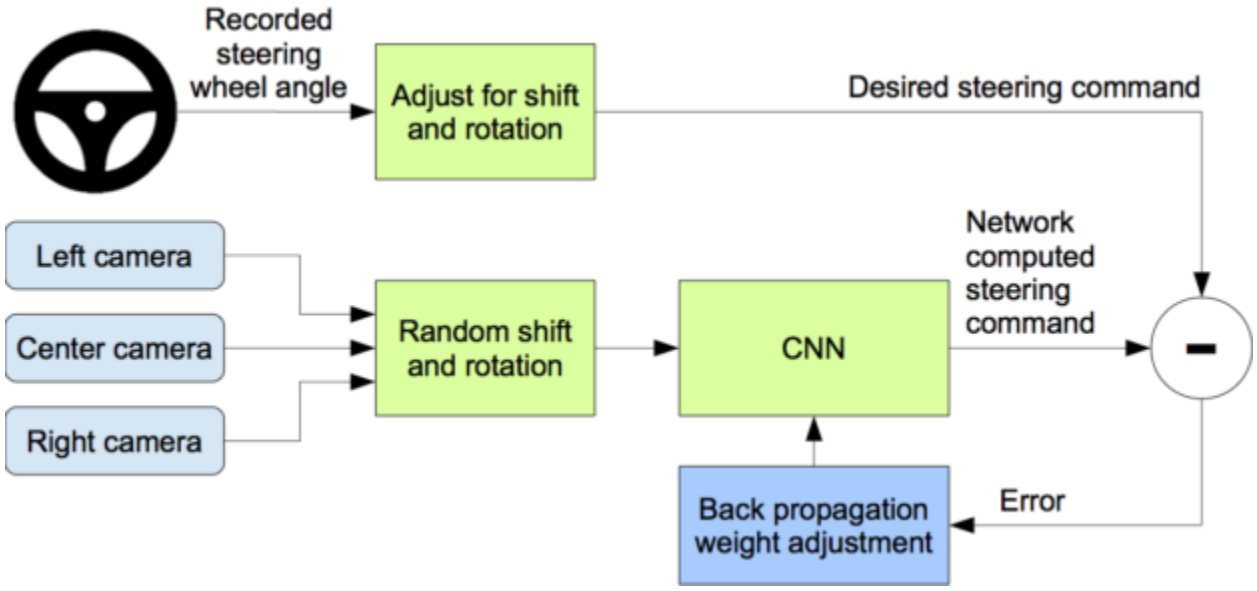


Fig.4.2 Training the Neural Network

Once trained, the network is able to generate steering commands from the video images of a single center camera. The below figure shows this configuration.

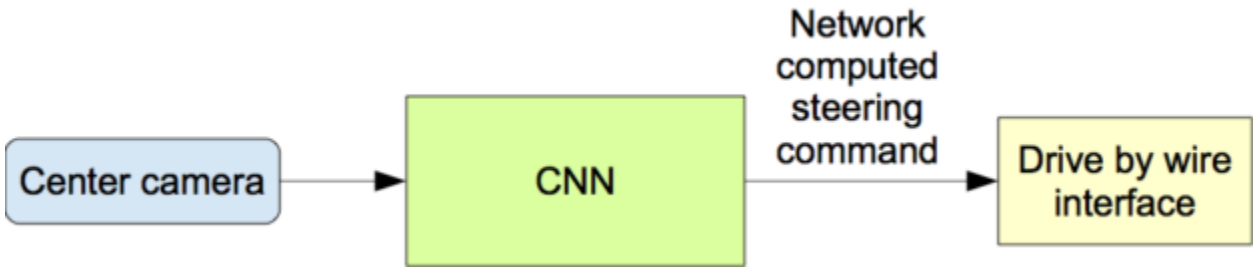


Fig.4.3 The trained network is used to generate steering commands from a single front-facing center camera.

4.3 NETWORK ARCHITECTURE

We train the weights of our network to minimize the mean-squared error between the steering command output by the network, and either the command of the human driver or the adjusted steering command for off-center and rotated images. Figure 5 shows the network architecture, which consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The input image is split into YUV planes and passed to the network.

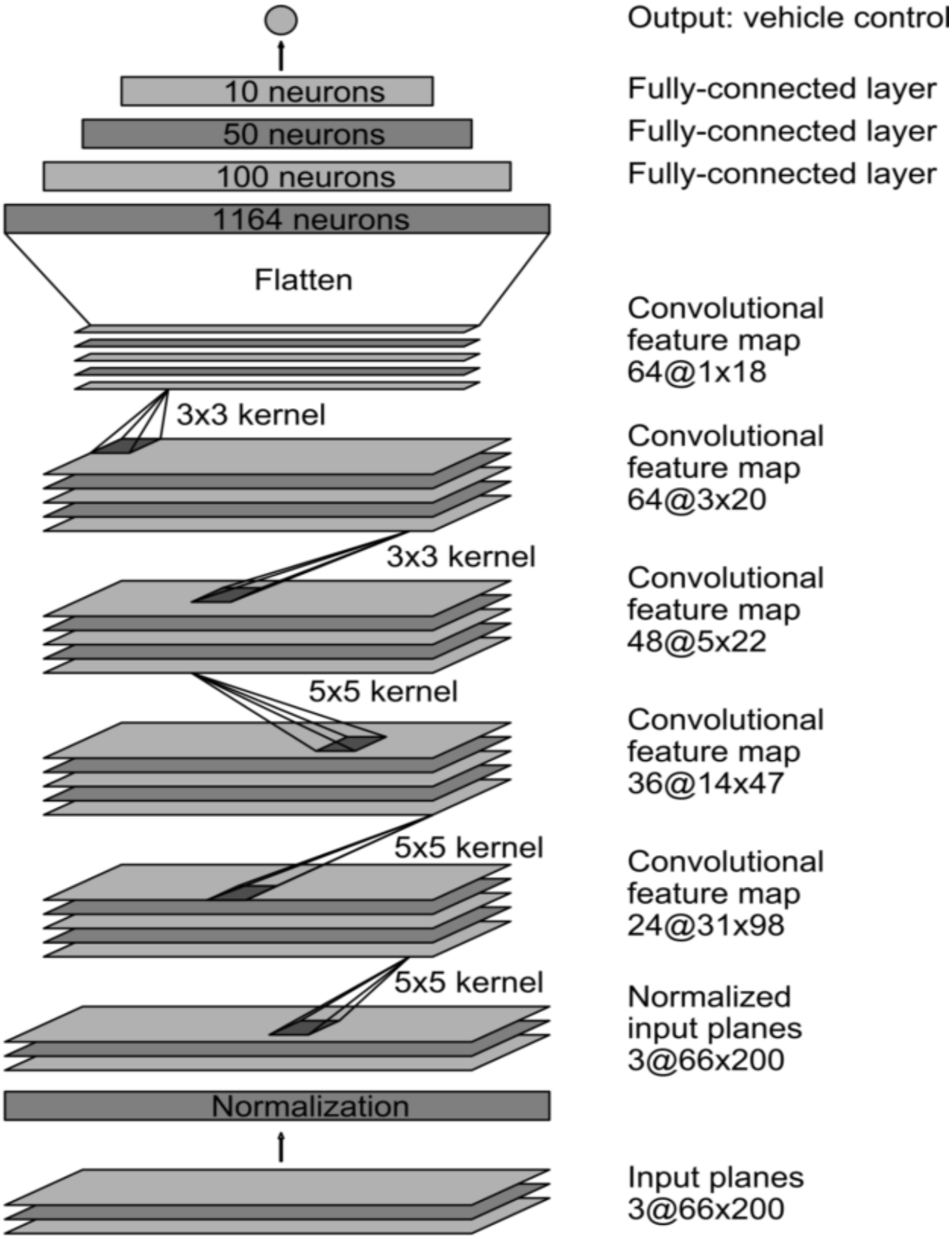


Fig.4.4 NVIDIA Model

The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture, and to be accelerated via GPU processing.

The convolutional layers are designed to perform feature extraction, and are chosen empirically through a series of experiments that vary layer configurations. We then use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel, and a non-strided convolution with a 3×3 kernel size in the final two convolutional layers.

We follow the five convolutional layers with three fully connected layers, leading to a final output control value which is the inverse-turning-radius. The fully connected layers are designed to function as a controller for steering, but we noted that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor, and which serve as controller.

4.4 UDACITY CAR SIMULATOR

Udacity recently made its self-driving car simulator source code available on their GitHub which was originally built to teach their Self-Driving Car Engineer Nanodegree students.
Now, anybody can take advantage of the useful tool to train your machine learning models to clone driving behavior.

You can manually drive a car to generate training data, or your machine learning model can autonomously drive for testing.

In the main screen of the simulator, you can choose a scene and a mode.

Fig.4.5 Udacity Car simulator start-up page

First, you choose a scene by clicking one of the scene pictures. In the above, the lake side scene (the left) is selected.

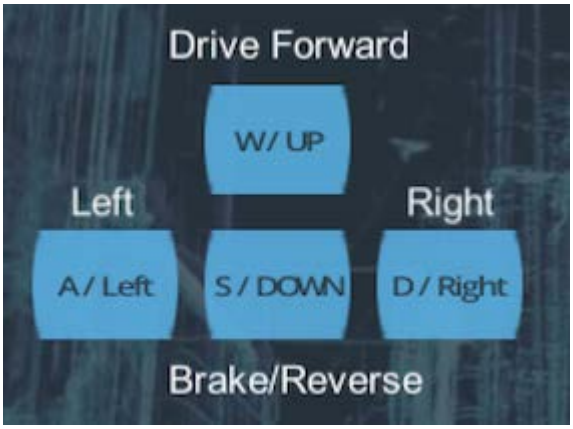
Next, you choose a mode: **Training Mode** or **Autonomous Mode**. As soon as you click one of the mode buttons, a car appears at the start position.

Fig.4.6 Simulator Train Model

4.4.1 Training Mode

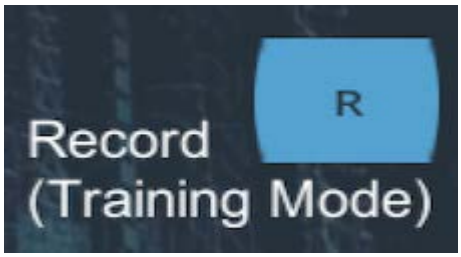
In the training mode, you drive the car manually to record the driving behavior. You can use the recorded images to train your machine learning model.

To drive the car, use the following keys:



If you like using mouse to direct the car, you can do so by dragging:

To start a recording your driving behavior, press R on your keyboard.



You can press R again to stop the recording.

Finally, use ESC to exit the training mode.

You can see the driving instructions any time by clicking CONTROLS button in the top right of the main screen.

4.4.2 Autonomous Mode

In the autonomous mode, you are testing your machine learning model to see how well your model can drive the car without dropping off the road / falling into the lake.

Technically, the simulator is acting as a server where your program can connect to and receive a stream of image frames from.

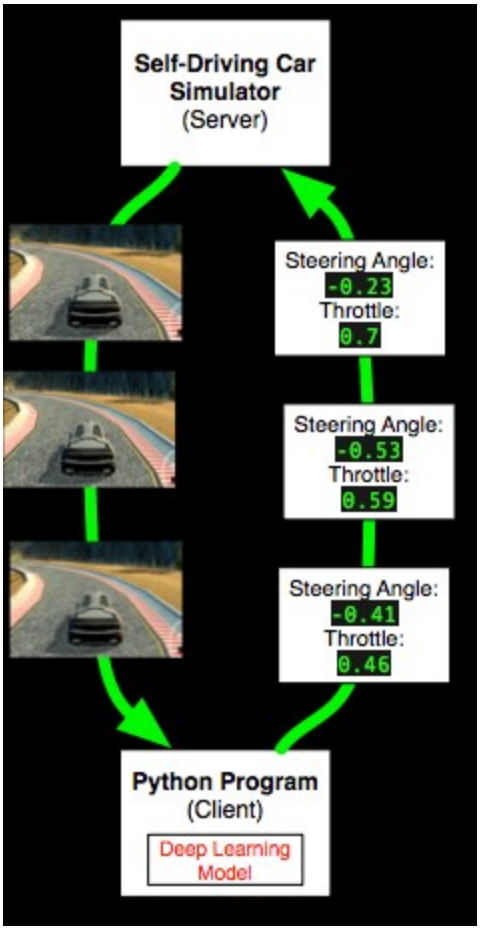


Fig.4.7 Simulator (Server) <-> Model (Client)

For example, your Python program can use a machine learning model to process the road images to predict the best driving instructions, and send them back to the server.

Each driving instruction contains a steering angle and an acceleration throttle, which changes the car's direction and the speed (via acceleration). As this happens, your program will receive new image frames at real time.

4.4.3 Model Architecture Design

The design of the network is based on the NVIDIA model, which has been used by NVIDIA for the end-to-end self-driving test. As such, it is well suited for the project.

It is a deep convolution network which works well with supervised image classification / regression problems. As the NVIDIA model is well documented, I was able to focus how to adjust the training images to produce the best result with some adjustments to the model to avoid overfitting and adding non-linearity to improve the prediction.

I've added the following adjustments to the model.

- I used Lambda layer to normalized input images to avoid saturation and make gradients work better.
- I've added an additional dropout layer to avoid overfitting after the convolution layers.
- I've also included ELU for activation function for every layer except for the output layer to introduce non-linearity.

In the end, the model looks like as follows:

- Image normalization
- Convolution: 5x5, filter: 24, strides: 2x2, activation: RELU
- Convolution: 5x5, filter: 36, strides: 2x2, activation: RELU
- Convolution: 5x5, filter: 48, strides: 2x2, activation: RELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: RELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: RELU
- Drop out (0.5)
- Fully connected: neurons: 100, activation: RELU
- Fully connected: neurons: 50, activation: RELU

- Fully connected: neurons: 10, activation: RELU
- Fully connected: neurons: 1 (output)

As per the NVIDIA model, the convolution layers are meant to handle feature engineering and the fully connected layer for predicting the steering angle. However, as stated in the NVIDIA document, it is not clear where to draw such a clear distinction. Overall, the model is very functional to clone the given steering behavior.

The below is a model structure output from the tensorflow.keras which gives more details on the shapes and the number of parameters.

Layer (type)	Output Shape	Params	Connected to
lambda_1 (Lambda)	(None, 66, 200, 3)	0	lambda_input_1
convolution2d_1 (Convolution2D)	(None, 31, 98, 24)	1824	lambda_1
convolution2d_2 (Convolution2D)	(None, 14, 47, 36)	21636	convolution2d_1
convolution2d_3 (Convolution2D)	(None, 5, 22, 48)	43248	convolution2d_2
convolution2d_4 (Convolution2D)	(None, 3, 20, 64)	27712	convolution2d_3
convolution2d_5 (Convolution2D)	(None, 1, 18, 64)	36928	convolution2d_4
dropout_1 (Dropout)	(None, 1, 18, 64)	0	convolution2d_5
flatten_1 (Flatten)	(None, 1152)	0	dropout_1
dense_1 (Dense)	(None, 100)	115300	flatten_1
dense_2 (Dense)	(None, 50)	5050	dense_1

Layer (type)	Output Shape	Params	Connected to
dense_3 (Dense)	(None, 10)	510	dense_2
dense_4 (Dense)	(None, 1)	11	dense_3
	Total params	252219	

Table. NVIDIA Model Summary

4.4.4 Data Preprocessing

Image Sizing

- the images are cropped so that the model won't be trained with the sky and the car front parts
- the images are resized to 66x200 (3 YUV channels) as per NVIDIA model
- the images are normalized (image data divided by 127.5 and subtracted 1.0). As stated in the Model Architecture section, this is to avoid saturation and make gradients work better)

4.4.5 Model Training

4.4.5.a Image Augmentation

For training, I used the following augmentation technique along with Python generator to generate unlimited number of images:

- Randomly choose right, left or center images.
- For left image, steering angle is adjusted by +0.2
- For right image, steering angle is adjusted by -0.2
- Randomly flip image left/right
- Randomly translate image horizontally with steering angle adjustment (0.002 per pixel shift)
- Randomly translate image vertically

- Randomly added shadows
- Randomly altering image brightness (lighter or darker)

Using the left/right images is useful to train the recovery driving scenario. The horizontal translation is useful for difficult curve handling (i.e. the one after the bridge).

4.4.5.b Examples of Augmented Images

Fig.4.8 Center Image

Fig.4.9 Left Image

Fig.4.10 Right Image

Fig.4.11 Flipped Image

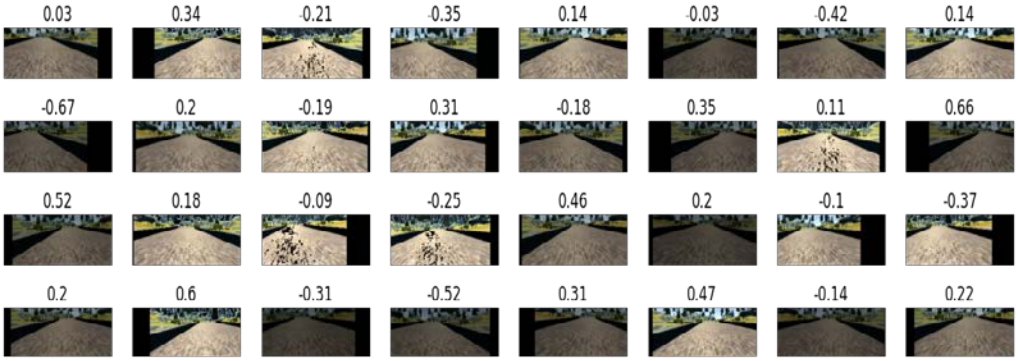


Fig.4.12 Sample Images from Dataset

4.4.5.b Training, Validation and Test

Splitted the images into train and validation set in order to measure the performance at every epoch. Testing was done using the simulator.

As for training,

- Used mean squared error for the loss function to measure how close the model predicts to the given steering angle for each image.
- Used Adam optimizer for optimization with learning rate of 1.0e-4 which is smaller than the default of 1.0e-3. The default value was too big and made the validation loss stop improving too soon.
- Used ModelCheckpoint from Keras to save the model only if the validation loss is improved which is checked for every epoch.

5. IMPLEMENTATION

5.1 HARDWARE SETUP

The motors, L298N board and the Raspberry Pi are fitted onto a chassis that has a movable axis in the front for steering. This is where the Pi Camera module and the servo motor are fitted. The DC motors are fitted at the back to drive the car. The L298N board is powered by a 11.1 V 2200mAh Lithium-Polymer battery and the Raspberry Pi is powered by a 10000mAh battery pack that outputs 5V@2A. Finally, the wheels are fitted to the motors and all components are connected with wires.

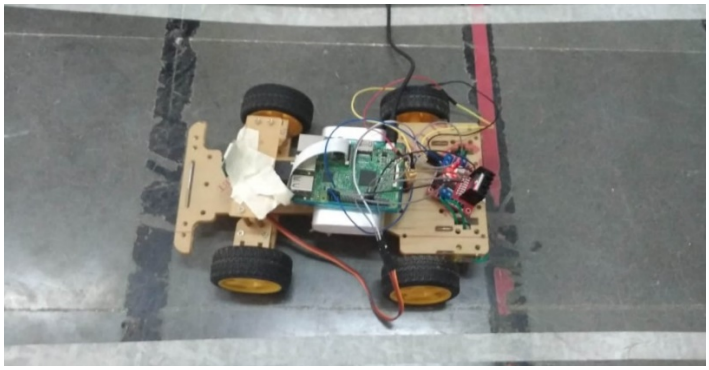


Fig.5.1 Our Self-Driving Car

5.2 TRAINING DATA COLLECTION

For training the neural network model that we built, we need training data. The input for the neural network is the image from the Pi Camera module and the output would be the steering angle. Since, the servo motor is controlled using PWM signals, instead of steering angle we are taking PWM signal value as the output. A single lane track is built with white tape as the boundary of the lane. The track is designed in such a way that there are ample amounts of both left and right turns. So, running the car both ways for 10 times generates sufficient data to train the neural network.

Fig.5.2 Track used for Training

The car is controlled manually with a self-made wired remote control and driven around the track. The remote control has 3 buttons – left, forward and right. Initially, the servo motor is calibrated to rotate left and right with PWM. With those values in mind the remote control is designed. On release of a button on the remote control will store the image from the Pi Camera module and save the PWM value, sent to the servo motor, at that moment in a list. On exiting the training program, a CSV file is generated with the image file names in a column and their corresponding PWM value in another column.

This way the car is driven 10 times on the track, both ways, to collect data for training the neural network. It was necessary to do it multiple times for two reasons:

- Neural networks are data hungry. More data equates to better performance of the neural network.
- It was also important to record left turns and right turns equal number of times to avoid biasing the neural network to any one turn.

	A	B	C
1		Image	Angle
2	0	test105/1.jpg	11.095245
3	1	test105/2.jpg	11.17876
4	2	test105/3.jpg	11.206715
5	3	test105/4.jpg	11.206715
6	4	test105/5.jpg	10.627355
7	5	test105/6.jpg	10.627355
8	6	test105/7.jpg	10.627355
9	7	test105/8.jpg	10.627355
10	8	test105/9.jpg	11.15797
11	9	test105/10.jpg	11.15797
12	10	test105/11.jpg	11.15797
13	11	test105/12.jpg	11.15797
14	12	test105/13.jpg	11.647065
15	13	test105/14.jpg	11.973905
16	14	test105/15.jpg	11.973905
17	15	test105/16.jpg	11.973905
18	16	test105/17.jpg	11.39927
19	17	test105/18.jpg	11.14033
20	18	test105/19.jpg	11.14033
21	19	test105/20.jpg	11.14033
22	20	test105/21.jpg	11.14033
23	21	test105/22.jpg	11.14033
24	22	test105/23.jpg	11.14033
25	23	test105/24.jpg	11.14033

Fig.5.3 Dataset Format (Sample)

5.3 TRAINING

5.3.1 Data Preprocessing

This step is essential to the training process as unwanted information can be eliminated and only useful data is used for training. Running the car around the track for 10 times, we have collected 2590 images. Then we randomly split 10% and labelled it as test data while the remaining 90% will be used for training.

The resolution of the Pi Camera module is set to (480,360). It is cropped to (480,310) to eliminate unnecessary information and only the track in front of the car remains in the image.

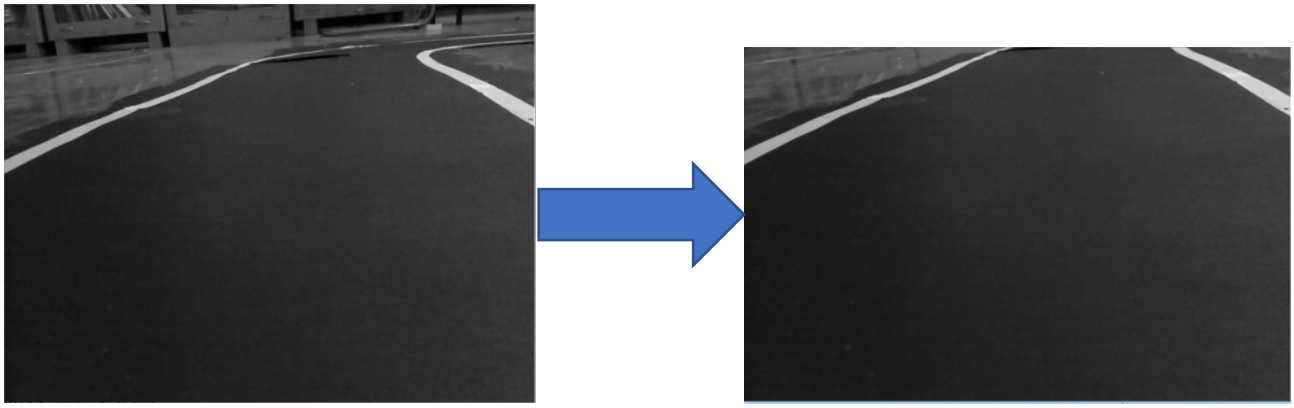


Fig.5.4 Before cropping and after cropping

For this application there is no use for a RGB image, so the RGB frames from the video stream are converted to grayscale.

Then the pixel values are normalized, i.e., divided by 255, so the pixel values range from 0 – 1

The PWM signal values are also normalized in a similar fashion and value lies between 0 and 1.

5.3.2 Neural Network Model

The below neural network architecture is inspired by the popular Nvidia model. We have modified the hyperparameters and omitted a couple of layers to suit our input image resolution.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 103, 160, 36)	360
max_pooling2d (MaxPooling2D)	(None, 51, 80, 36)	0
conv2d_1 (Conv2D)	(None, 17, 26, 48)	15600
max_pooling2d_1 (MaxPooling2D)	(None, 8, 13, 48)	0
conv2d_2 (Conv2D)	(None, 2, 4, 64)	27712
dropout (Dropout)	(None, 2, 4, 64)	0
flatten (Flatten)	(None, 512)	0
dense (Dense)	(None, 100)	51300
dense_1 (Dense)	(None, 50)	5050
dense_2 (Dense)	(None, 10)	510
dense_3 (Dense)	(None, 1)	11

Total params: 100,543

Trainable params: 100,543

Non-trainable params: 0

5.3.3 Training Process

The optimizer used is ADAM and it is used to optimize the loss function – Mean Square Error (MSE). The model is initialized and trained for 40 epochs. And is simultaneously validated against the test data every epoch.

```
Epoch 34/40
2331/2331 [==============================] - 7s 3ms/sample - loss: 0.0044 - acc: 0.0884 -
val_loss: 0.0102 - val_acc: 0.0811
Epoch 35/40
2331/2331 [==============================] - 7s 3ms/sample - loss: 0.0043 - acc: 0.0884 -
val_loss: 0.0094 - val_acc: 0.0811
Epoch 36/40
2331/2331 [==============================] - 7s 3ms/sample - loss: 0.0040 - acc: 0.0884 -
val_loss: 0.0101 - val_acc: 0.0811
Epoch 37/40
2331/2331 [==============================] - 8s 3ms/sample - loss: 0.0043 - acc: 0.0884 -
val_loss: 0.0107 - val_acc: 0.0811
Epoch 38/40
2331/2331 [==============================] - 8s 3ms/sample - loss: 0.0041 - acc: 0.0884 -
val_loss: 0.0101 - val_acc: 0.0811
```

Fig. Screenshot of the training process

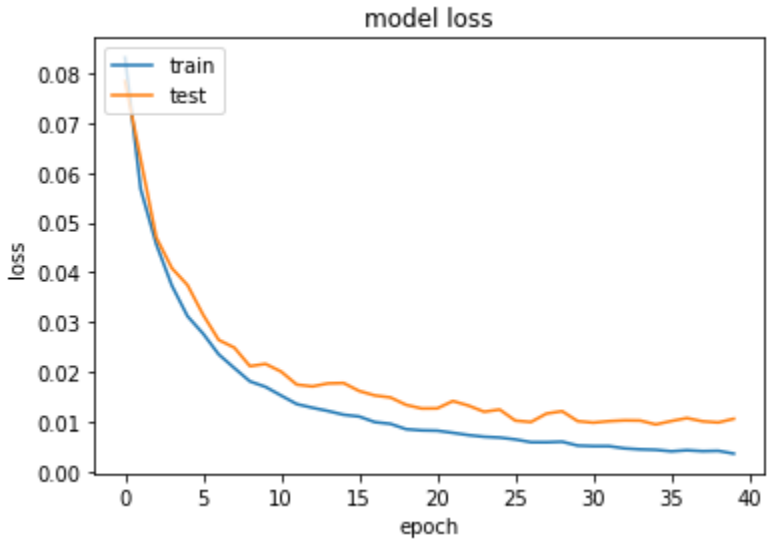


Fig.5.5 Plot of train and test loss vs number of epochs

6. RESULTS

```
Epoch 40/40
2331/2331 [==============================] - 7s 3ms/sample - loss: 0.0036 - acc: 0.0884 -
val_loss: 0.0106 - val_acc: 0.0811
```

After training the model for 40 epochs, the model achieved training loss – 0.0036 and validation loss – 0.0106.

Post training the weights of the model are saved so that they can be loaded and used in future. This model file (.h5) is copied to the Raspberry Pi and loaded into the model. Now, the car is ready and is put to test on the track.

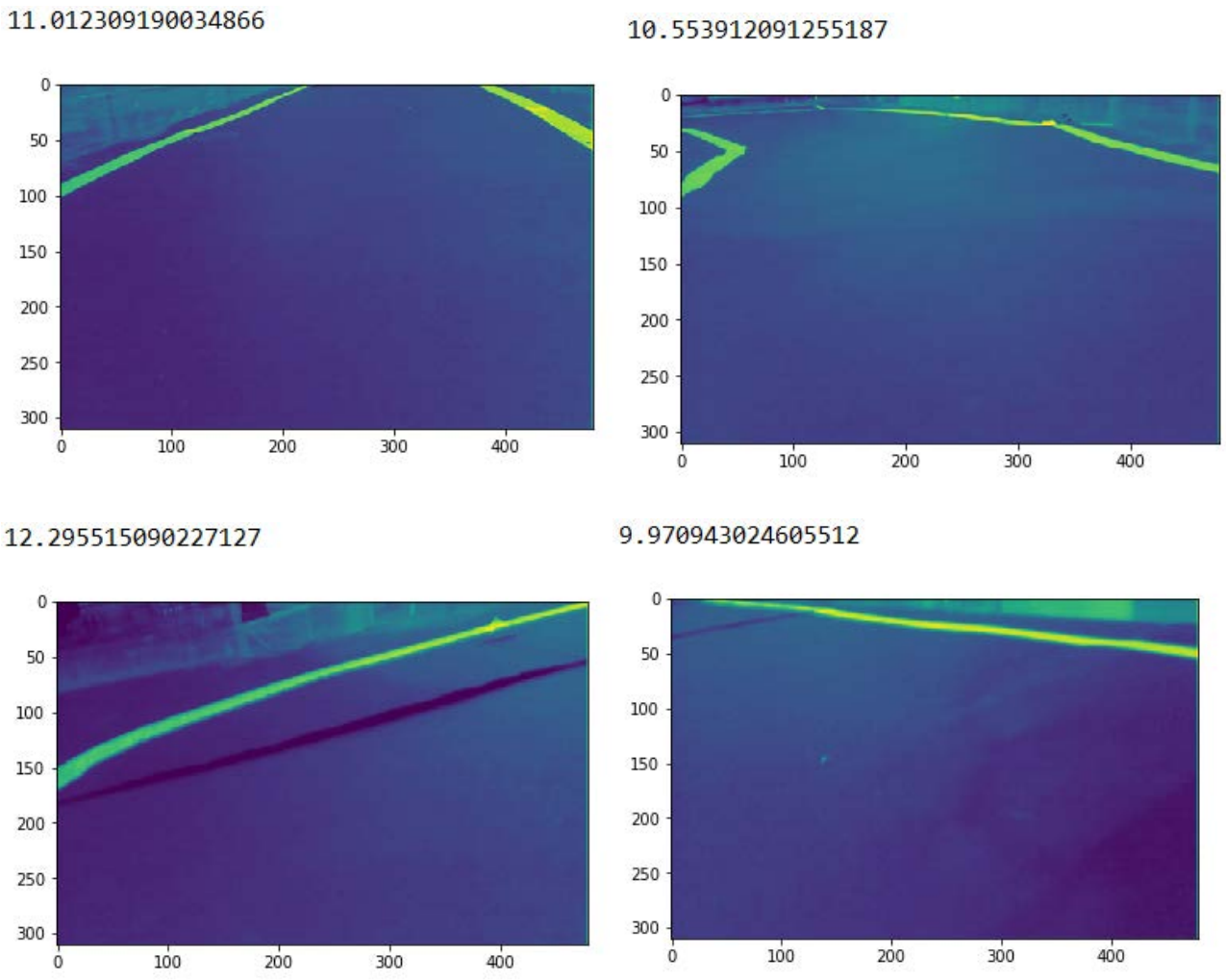


Fig.5.6 Output of the model running on Raspberry Pi

7. CONCLUSION AND FUTURE SCOPE

We have implemented lane keeping by running the trained CNN-model on the Raspberry Pi that is connected to the servo motor, DC motors and the Raspberry Pi Camera module. The shortest path generation and localization were not integrated with the car but implemented separately as the processing capability of Raspberry Pi was not sufficient to run these modules simultaneously.

The model's performance is good enough for a small track like ours but is still susceptible to lighting changes and reflections. So, collecting more data that includes these conditions will help the model perform well in all cases. The Pi Camera Module that is used in this project is set to a resolution of 480x360. Capturing and using higher resolution images as training data improves the performance of the model. Wide-angle cameras also help for better performance as more data is captured in a single image than a regular camera. But increasing resolution of the images also increases the computation complexity. While running the model on the Raspberry Pi, to match the processing time, the speed of the car had to be reduced so that the Raspberry Pi doesn't skip frames and predict wrong steering angle. Better CPUs and GPUs can perform computations faster, that enables the car to move faster and also other features can also be added. Object detection can be integrated that can be used to detect traffic signs, traffic lights, people, other vehicles, etc. and move accordingly. Also, sensors like LIDAR, ultrasonic sensors, RADARs and use of GPS can make the car aware of its surroundings. Then, integrating these sensors with Sensor Fusion can improve the system performance by correcting deficiencies of individual sensors to calculate accurate position and orientation.

The same is with the real-world self-driving cars. The performance can go down with different lighting, reflections on the road and weather conditions. To ensure good performance at all times, the model has to be trained with all such cases. The training data should contain all such cases and in large volume. But collecting data is a tedious job. So, GANs can be used to generate such data, different weather and lighting conditions, using the images taken in normal conditions.

References

1. https://towardsdatascience.com/machine-learning-for-beginners-d247a9420dab

2. https://neurohive.io/en/popular-networks/vgg16/

3. https://devblogs.nvidia.com/deep-learning-self-driving-cars/

4. Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backprop- agation applied to handwritten zip code recognition. Neural Computation, 1(4):541–551, Winter 1989. URL: http://yann.lecun.org/exdb/publis/pdf/lecun-89e.pdf.

5. Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/ 4824-imagenet-classification-with-deep-convolutional-neural-networks. pdf.

6. L. D. Jackel, D. Sharman, Stenard C. E., Strom B. I., , and D Zuckert. Optical character recognition for self-service banking. AT&T Technical Journal, 74(1):16–24, 1995.

7. Large scale visual recognition challenge (ILSVRC). URL: http://www.image-net.org/ challenges/LSVRC/.

8. Net-Scale Technologies, Inc. Autonomous off-road vehicle control using end-to-end learning, July 2004. Final technical report. URL: http://net-scale.com/doc/net-scale-dave-report.pdf.

9. Dean A. Pomerleau. ALVINN, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University, 1989. URL: http://repository.cmu.edu/cgi/viewcontent. cgi?article=2874&context=compsci.

10. Danwei Wang and Feng Qi. Trajectory planning for a four-wheel-steering vehicle. In Proceedings of the 2001 IEEE International Conference on Robotics & Automation, May 21–26 2001. URL: http: //www.ntu.edu.sg/home/edwwang/confpapers/wdwicar01.pdf.

11. https://github.com/naokishibuya/car-behavioral-cloning

12. https://towardsdatascience.com/introduction-to-udacity-self-driving-car-simulator-4d78198d301d

13. https://www.d2l.ai/

14. https://www.deeplearningbook.org/

15. https://opencv.org/about/

16. https://www.python.org/downloads/release/python-365/

17. https://www.raspbian.org/RaspbianFAQ