EasyChair Preprint

№ 1248

Driverless Car - Design of a Parallel and Self-Organizing System

Poondru Prithvinath Reddy

June 30, 2019

DRIVERLESS CAR - DESIGN OF A PARALLEL AND SELF – ORGANIZING SYSTEM.

POONDRU PRITHVINATH REDDY

ABSTRACT

For autonomous vehicles, several real-time systems must work tightly together. These real-time systems, include environment mapping and understanding, localization, route planning and movement control. All these real-time systems work simultaneously and use artificial neural networks which are self-organizing systems. The self-driving car itself needs to be equipped with the appropriate computational hardware such as parallel computing power of modern graphics processors and software infrastructure for supporting implementation of DNN & CNNs. There are two approaches for applying deep learning in self-driving cars. The first one is semantic abstraction and the second is end-to-end learning system. Our chosen approach is semantic abstraction where the problem of autonomous driving is broken down into several components and at the end, these components are glued together into master network that makes the driving decisions. Also implementation of image classification in traffic signs dataset using Deep Neural Network with TensorFlow is presented.

INTRODUCTION

A **self-driving car**, also known as a **robot car**, **autonomous car**, or **driverless car**, is a vehicle that is capable of sensing its environment and moving with little or no human input. Autonomous cars combine a variety of sensors to perceive their surroundings, such as, Lidar*,* radar, sonar, GPS, Odometry and inertial measurement units. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles .

CONCEPT OF AUTONOMOUS DRIVING

A car capable of autonomous driving should be able to drive itself without any human input To achieve this, the autonomous car needs to sense its environment, navigate and react without human interaction. A wide range of sensors, such as LIDAR, RADAR, GPS, wheel odometry sensors and cameras are used by self-driving cars to perceive their surroundings. In addition, the autonomous car must have a control system that is able to understand the data received from the sensors and make a difference between

traffic signs, obstacles, pedestrian and other expected and unexpected things on the road .

For a machine to be called a robot, it should satisfy at least three important capabilities: to be able to sense, plan, and act . For a car to be called an autonomous car, it should satisfy the same requirements . Self-driving cars are essentially robot cars that can make decisions about how to get from point A to point B.

PARALLEL SYSTEMS

Parallel Systems are designed to speed up the **execution** of programs by dividing the program into multiple fragments and **processing** these fragments simultaneously. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

Parallel computing is an evolution of serial computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs.

SELF – ORGANIZING SYSTEM

The term Self - Organizing Systems refers to a class of systems that are able to change their internal structure and their function in response to external circumstances.By self-organization it is understood that elements of a system are able to manipulate or organize other elements of the same system in a way that stabilizes either structure or function of the whole against external fluctuations.

Self-organization is defined as a process by which systems that are in general composed of many parts spontaneously acquire their structure or function without specific interference from an agent that is not part of the system.

Self-organizing systems are dynamic, non-deterministic, open, exist far from equilibrium.Often, they are characterized by multiple time-scales of their internal and / or external interactions, they possess a hierarchy of structural and / or functional levels and they are able to react to external input in a variety of ways.Many self-organizing systems are non-teleological, i.e . they do not have a specific purpose except their own existence.As a consequence, self-maintenance is an important function of many self-organizing systems . Most of these systems are complex and use reduncancy to achieve resilience.

There are numerous examples of man-made systems or systems which involve man that exhibit self-organization phenomena.Here we shall discuss a few examples from various areas,trafic infrastructure,self-organizing neural networks and the development of the Internet.All of these examples deal with the trans-portation of matter, energy or information in networks.

Artificial neural networks as self-organized systems

As already mentioned,natural neural connection patterns in brains exhibit self-organized structures. Self-organization phenomena can be found everywhere in the inanimate and animate world. We provide a particularly interesting example, namely *self-organization phenomena of the human brain*. The human brain is the most complex system we know in the world. It is composed of up to 100 billions neurons which are strongly interconnected. For instance, a single neuron can have more than 10,000 connections to other neurons. The central question is: who or what steers the numerous neurons so that they can produce macroscopic phenomena such as the coherent steering of muscles in locomotion, grasping, vision i.e. in particular pattern recognition, decision making etc

SELF – DRIVING VEHICLES

The following sensors should be present in all self-driving cars:

Global positioning system (GPS). Global positioning system is used to determine the position of a self-driving car by triangulating signals received from GPS satellites . It is often used in combination with data gathered from an IMU and wheel odometry encoder for more accurate vehicle positioning and state using sensor fusion algorithms.

Light detection and ranging (LIDAR). A core sensor of a self-driving car, this measures the distance to an object by sending a laser signal and receiving its reflection . It can provide accurate 3D data of the environment, computed from each received laser signal. Self-driving vehicles use LIDAR to map the environment and detect and avoid obstacles .

Camera. Camera on board of a self-driving car is used to detect traffic signs, traffic lights, pedestrians, etc. by using image processing algorithms .

RADAR. RADAR is used for the same purposes as LIDAR. The advantages of RADAR over LIDAR are that it is lighter and has the capability to operate in different conditions .

Ultrasound sensors. Ultrasound sensors play an important role in the parking of self-driving vehicles and avoiding and detecting obstacles in blind spots, as their range is usually up to 10 metres .

Wheel odometry encoder. Wheel encoders provide data about the rotation of car's wheels per second. Odometry makes use of this data, calculates the speed, and estimates the car's position and velocity based on it.

Inertial measurement unit (IMU). An IMU consists of gyroscopes and accelerometers. These sensors provide data on the rotational and linear motion of the car, which is then used to calculate the motion and position of the vehicle regardless of speed .

On-board computer. This is the core part of any self-driving car. As any computer, it can be of varying power, All sensors connect to this computer, which has to make use of sensor's data by understanding it, planning the route and controlling the car's actuators. The control is performed by sending the control commands such as steering angle, throttle and braking to the wheels, motors and servo of the autonomous car .

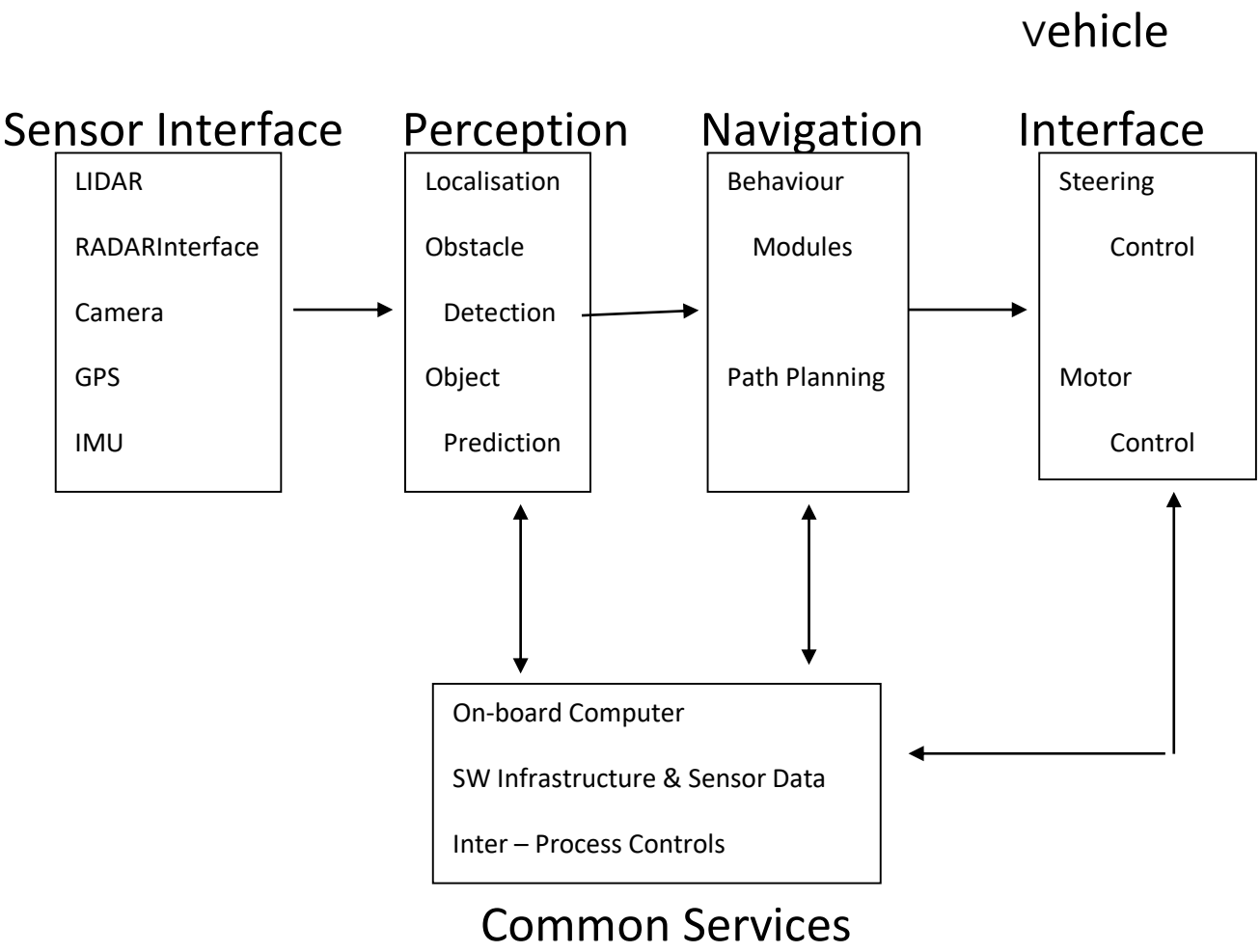

Figure 1 illustrates the SW block diagram of the standard self-driving car.

Each block seen in Figure 1 can interact with other blocks using inter-process communication (IPC) and identified the following blocks for the SW block diagram of a typical self-driving car:

Sensor interface modules. All communication between sensors and the car is performed in this block, as it enables data acquired from sensors to be shared with other blocks.

Perception modules. These modules process perception data from sensors such as LIDAR, RADAR and cameras, then segment the processed data to locate different objects that are staying still or moving.

Navigation modules. Navigation modules determine the behaviour of the self-driving car, as they have route and motion planners, as well as a state machine of car's behaviour .

Vehicle interface. This interface's goal is to send control commands such as steering, throttle and braking to the car after the path has been plotted in the navigation module.

Common services. Common services module controls the car's SW reliability by allowing logging and time-stamping of car's sensor data.

Software Modules for Autonomous Driving

Predetermined shape and motion descriptors are programmed into the system to help the car make intelligent decisions. For instance, if the car detects a 2 wheel object and determines the speed of the object as 10mph rather than 50 mph, the car instantly interprets that this vehicle is a bicycle and not a motorbike and behaves accordingly. Several such programs fed into the car's central processing unit will work simultaneously, helping the car make safe and intelligent decisions on busy roads.

At the moment, before a self-driven car is tested, a regular car is driven along the route and maps out the route and it's road conditions including poles, road markers, road signs and more. This map is fed into the car's software helping the car identify what is a regular part of the road. As the car moves, its Velodyne laser range finder kicks in and generates a detailed 3D map of the environment at that moment. The car compares this map with the pre-existing map to figure out the non-standard aspects in the road, rightly identifying them as pedestrians and/or other motorists, thus avoiding them.

While the vehicle does slow down to allow other motorists to go ahead, especially in 4 way intersections, the car has also been programmed to advance ahead if it detects that the other vehicle is not moving.

The main task faced by driverless cars software developers is to make the product that will adapt to external environmental factors as quickly as possible.

Self-Driving car key functions are – HD_Maps, Route_Planning, Detect_Obstacles, Avoid_Obstacles, Detect_Traffic_Signs, Detect_Traffic_Lights, Detect_Pedestrians, Distace_Perception, Detect_Road_Edge_Stone, Detect_Road_Markings, Detect_Poles, Detect_Other_Motorists, Object_Tracker (velocity and attitude), and Object_Predictor

Perception

The perception capability of Autonomous Car is composed of Localization, Detection, and Prediction. Detection uses cameras and LiDARs with sensor fusion algorithms and deep neural networks. Prediction is based on the results of Localization and Detection. Localization is achieved by 3D maps and SLAM algorithms.

object_detector reads image data from cameras, and provides image-based object detection capabilities. Multiple classes of detection are supported, such as cars and passengers.

object_tracker predicts the motion of objects detected and identified by the above packages. The result of Prediction is based on the results of Localization and Detection. Further it is also used for prediction of the object behavior and estimation of the object velocity.

Prediction

object_predictor uses the result of object tracking described above to predict the future trajectories of moving objects, such as cars and passengers.

collision_predictor uses the result of object_predictor to predict if the vehicle is involved in possible collision against the moving objects. The waypoint and the velocity information of the vehicle is also required as input data in addition to the result of object tracking.

Planning

The last piece of computing in Autonomous Car is a planning module. The role of this module is to make plans of global mission and local motion based on the results of the perception and the decision modules. For example, the velocity of the vehicle is planned to become zero in front of an object with a safety margin or at a stop line if the state of vehicle is set to "stop". Another example is that the trajectory of the vehicle is planned to bypass an obstacle if the state of vehicle is set to "avoid". The primary packages included in the planning module are the following.

route_planner searches for a global route to the destination. The route is represented by a set of intersections in the road network.

lane_planner determines which lanes to be used along with the route published by route_planner. The lanes are represented by an array of waypoints, i.e., multiple waypoints, each of which corresponds to a single lane.

Motion

velocity_planner updates a velocity plan on the waypoints subscribed from lane_planner, so as to speed down/up against surrounding vehicles and road features such as stop lines and traffic lights.

Actuation

The computational output of Autonomous Car is a set of velocity, angular velocity, wheel angle, and currvature. These pieces of information are sent as commands to the controller through the vehicle interface. Controlling the steering and throttle needs to be taken care of by the controller.

Why ROS is interesting for Autonomous Cars

Robot Operating System (ROS) is a mature and flexible framework for robotics programming. ROS provides the required tools to easily access sensors data, process that data, and generate an appropriate response for the motors and other actuators of the robot.

ROS is interesting for autonomous cars because:

There is a lot of code for autonomous cars already created. Autonomous cars require the creation of algorithms that are able to build a map, localize the robot using lidars or GPS, plan paths along maps, avoid obstacles, process pointclouds or cameras data to extract information, etc… All kind of algorithms required for the navigation of wheeled robots is almost directly applicable to autonomous cars. Hence, since those algorithms have already been created in ROS, self-driving cars can just make use of them off-the-shelf.

Visualization tools already available. ROS has created a suite of graphical tools that allow the easy recording and visualization of data captured by the sensors, and represent the status of the vehicle in a comprehensive manner. Also, it provides a simple way to create additional visualizations required for particular needs. This is tremendously useful when developing the control software and trying to debug the code.

It is relatively simple to start an autonomous car project with ROS onboard.

The ROS platform could greatly shorten the robot development cycle, and simultaneous localisation and mapping (SLAM) could easily be realised using ROS . This is possible

because ROS already has ready packages for this purpose called gmapping. By using this package, ROS-based self-driving car could simply map the environment by using LIDAR sensor .

NEURAL NETWORKS FOR AUTONOMOUS DRIVING

There are various tasks that can be solved by DNNs(Deep Neural Networks) that are useful for autonomous driving, but the four fundamental tasks are: Classification, detection, segmentation and regression.

Other more advanced tasks like scene understanding or path planning build up on those basic four. Classification networks identify and categorize objects. A vision classifier network for example categorizes objects in a picture frame.

Networks with detection tasks in contrast are able to recognize and mark certain objects in a frame. Networks with segmentation tasks partition pictures into sets of pixels (segments) to locate boundaries of objects. For this task special CNNs(convolutional neural networks) with Encoder-Decoder architectures are usable . Finally regression tasks are often solved in the last layer of a network to map a continuous inputs to continuous outputs.

There are in general four questions a car needs to be able to answer to achieve the final goal of autonomy.

1) Where am I? →Localization and Mapping

2) Where is everybody else? →Scene Understanding

3) How do I get from A to B? →Movement Planning

4) What are the obstructions? →Detection

Answering those questions can be realized in two different ways. One way is via semantic abstraction where each task is executed in a seperate network and afterwards combined with classical control & decision-making algorithms . The other approach is called end-to-end, where a single DNN takes all the car's inputs and computes a final control command as output. It is important to notice that some applications cannot be assigned to only one specific task. Therefore some of the following applications overlap in their topics.

A. Detection and Classification

One of the first autonomous driving tasks mastered by DNNs was traffic sign recognition. In fact CNNs are since 2012 better than humans on recognizing street

signs with an accuracy of 99,46% . Related topics like line, traffic light and vehicle detection have accuracies on a similar level when applied on state-of-art CNN architectures . An example of a state-of-the-art CNN for detection and localization tasks, developed is YOLO Darknet v2. It can detect more than 9000 Objects in real-time at 40- 70 fps with a mean accuracy of nearly 80%, which makes it capable of detecting everything necessary for automotive tasks in a video or an onboard-camera.

B. Scene Understanding

Semantic segmentation is a technique used for road scene understanding. and use a special CNN encoder-decoder architecture . After the input image is processed through the network a pixel wise classification is computed to identify each pixel to the belonging object. It achieves a prediction accuracy of around 88% for cars and 96% for roads. Although it struggles with pedestrians, the achieved accuracy of 62% still outperforms all other tested algorithmic methods by over 10%. Surround Vehicle Trajectory Analysis (SVTA) is using Long Short Term Memory (LSTM) in RNNs (Recurrent Neural NETWORKS) as well as 3D trajectory cues. The same problem is faced when future predictions want to be made about what other road users are up to do. The sensor signals are fed into a RNN-LSTM network to predict the trajectories of surrounding vehicles. It is concluded that the system was able to make good predictions for coarse labels such as turning versus going straight but predicting a finer activity label space with more output options was problematic.

C. Localization and Mapping

Using the camera signal to get accurate bounding box locations around pixels of detected objects also the distance and relative speed is obtainable by matching with the radar signal . Besides 2D, also 3D object detection is possible from single monocular images that objects recognized by the vehicle's sensors should be on the ground plane (zero height). Chenyi Chen et al used this assumption to estimate car distances . Like the SVTA sytsm, the camera and lidar signals of the KITTI dataset served as input. For this approach a two CCN system was used. One for close range (2-25m) and one for far range (15-55m) object detection due to the low resolution of the input images. For the final distance projection the output of both CNNs are combined.

D. Movement Planning

Another Application is movement planning on small scales like finding a way around obstacles using short range sensors like camera, lidar, sonar and radar and navigation on the bigger scale with long range sensors like GPS where finding the fastest or most efficient route is important. Huang et al. developed a framework visual path prediction. It consists of two CNNs that separately model the spatial and temporal context. Drive.ai

let’s their small fleet of four autonomous Audis even take one further step. Their cars do decision making and motion planing on difficult situations like the American four way stop, where the first come first serve rule is applied, or even turning on red, which is allowed in most intersections.

SEMANTIC ABSTRACTION VERSUS END –TO – END DEEP LEARNING

But in what ways is deep learning specifically applied in self-driving cars? There are two main approaches, which both have their own advantages and shortcomings.

The first one is using semantic abstraction, where the problem of autonomous driving is broken down into several components. These are algorithms that are focused only on one part of the task. For example, one component could be focused on pedestrian detection, another to detecting lane markings and a third one to detecting objects beyond the lanes. At the end, these components are “glued together” into a master network that makes the driving decisions. On the other hand, a network can be constructed that detects and classifies multiple classes or even does semantic segmentation.

The advantages of such a system, is the lower tolerance for mistakes, the ability to pinpoint the errors more easily and the capability to manage unpredictable situation better. Its shortcomings, however, are also big, since it requires huge pre-work and complex programming .

The second approach is the more “disruptive” end-to-end learning approach. This is where the car actually teaches itself how to drive, based on a huge set of human driving data. Although this approach also has big shortcomings, such as the requirement of having a huge training data set and the difficulty to be trained and tuned properly, it is very promising for the future of intelligent vehicles.

As noted before, an end-to-end learning system especially, requires to be fed a huge amount of training data, in order to predict as many driving scenarios as possible and to fulfil a minimum safety requirement.

In this paper our approach is using semantic abstraction, where the problem of autonomous driving is broken down into several components. These are algorithms that are focused only on one part of the task. At the end, these components are “glued together” into a master network that makes the driving decisions.

PARALLEL FUNCTIONS IN AUTONOMOUS DRIVING

Autonomous driving is extremely complex and poses challenging problems and requires use of powerful and energy-efficient computer systems that employ several types of processor. Central processing units exist alongside graphics controllers and deep learning accelerators. Highly automated driving functions are not possible without the parallel computing power of modern graphics processors and Graphics processors are replacing CPUs in automated vehicles.

Vehicle's Location and Environment

> 3d image processing with artificial neural networks

> Multiprocessor graphics hardware (GPUs)

Prediction & Decision algorithms

> artificial neural networks

> specialized multiprocessor hardware

> early, independent hardware validation

High accuracy, real-time MAPs

> environmental / spatial modeling

> simultaneous localization and mapping (SLAM)

Detect and Avoid Obstacles

Simultaneous Interpretation of Predetermined Shapes and Motion Descriptors

Route Planning

IMAGE CLASSIFICATION USING ARTIFICIAL NEURAL NETWOKS

This is an implementation of one of key tasks of Autonomous Driving.

This example shows how to build a deep neural network and also to train, evaluate and optimize it with TensorFlow.

Image classification versus object detection

Often people confuse image classification and object detection scenarios. In general, if we want to classify an image into a certain category, we use image classification. On the other hand, if we aim to identify the location of objects in an image, and count the number of instances of an object, we can use object detection.

With an image classification model, we generate image features (through traditional or deep learning methods) of the full image. These features are aggregates of the image. With object detection, we do this on a more fine-grained, granular, regional level of the image.

Deep learning is a subfield of machine learning that is a set of algorithms that is inspired by the structure and function of the brain.

TensorFlow is the machine learning framework that Google created and used to design, build, and train deep learning models.

The following steps will involve in performing deep learning :

> We load in data on Belgian traffic signs and explore it with simple statistics and plotting.

> There is a need to change the data in such a way that we can feed it to the model. That's why we'll rescale the images and convert them to grayscale.

> Next, we finally get started on NN Model and We'll build up the model layer per layer;

> Once the architecture is set up, we use it to train the NN model and to also evaluate the model by feeding some test data to it.

We used 62 images of different traffic signs from Belgian Traffic Signs dataset. Let us download the Belgian Traffic Signs dataset from https://btsd.ethz.ch/shareddata/. We get the two zip files listed next to "BelgiumTS for Classification (cropped images), which are called "BelgiumTSC_Training" and "BelgiumTSC_Testing". We'll see that the testing, as well as the training data folders, contain 61 subfolders, which are the 62 types of traffic signs that we'll use for classification . Additionally, we'll find that the files have the file extension .ppm or Portable Pixmap Format.

Let's get started with importing the data into our workspace. Let's start with the User-Defined Function (UDF) load_data():

We start with a pretty simple analysis with the help of the ndim and size attributes of the images array: Note that the images and labels variables are lists, so we might need to use np.array() to convert the variables to an array in our own workspace. Next, we can also take a look at the distribution of the traffic signs: We clearly see that not all types of traffic signs are equally represented in the dataset. At first sight, we see that there are labels that are more heavily present in the dataset than others: example the

labels 22, 32, 38, and 61 . But when the data mostly consists of images, the step that one should take to explore the data is by visualizing it.

Let's check out some random traffic signs:

First, make sure that we import the pyplot module of the matplotlib package under the common alias plt.

Then, we're going to make a list with 4 random numbers. These will be used to select traffic signs from the images array that we have just inspected in the previous section. In this case, we go for 300, 2250, 3650 and 4000.

Next, we'll say that for every element in the length of that list, so from 0 to 4, we're going to create subplots without axes . In these subplots, we're going to show a specific image from the images array that is in accordance with the number at the index i. In the first loop, you'll pass 300 to images[], in the second round 2250, and so on. Lastly, we'll adjust the subplots so that there's enough width in between them. As guessed the 62 labels that are included in this dataset, the signs are different from each other. Also These four images are not of the same size!

Let's start first with extracting some features - we'll rescale the images, and we'll convert the images that are held in the images array to grayscale. We'll do this color conversion mainly because the color matters less in classification questions . For detection, however, the color does play a big part! So in those cases, it's not needed to do that conversion!

To tackle the differing image sizes, we're going to rescale the images; We can do this with the help of the skimage or Scikit-Image library, which is a collection of algorithms for image processing.

In this case, the transform module will come in handy, as it offers a resize() function; We'll see that we make use of list comprehension to resize each image to 28 by 28 pixels. Once again, for every image that we find in the images array, we'll perform the transformation operation that is borrowed from the skimage library. Finally, we store the result in the images28 variable:

We can check the result of the rescaling operation by re-using the code to plot the 4 random images with the help of the traffic_signs variable. But don't forget to change all references to images to images28.

As said in the introduction , the color in the pictures matters less when we're trying to answer a classification question. That's why we'll also go through the trouble of converting the images to grayscale.

Just like with the rescaling, we again count on the Scikit-Image library to help out; In this case, it's the color module with its rgb2gray() function that we need to use to get where we need to be.

However, don't forget to convert the images28 variable back to an array, as the rgb2gray() function does expect an array as an argument.

Double check the result of grayscale conversion by plotting some of the images;

Deep Learning With TensorFlow

Now that we have explored and manipulated the data, it's time to construct neural network architecture with the help of the TensorFlow package!

Modelling The Neural Network

It's time to build up our neural network, layer by layer.

Import tensorflow into our workspace under the conventional alias tf. Then, we can initialize the Graph with the help of Graph(). We use this function to define the computation.

In this case, we set up a default context with the help of as_default(), which returns a context manager that makes this specific Graph the default graph. We use this method if we want to create multiple graphs in the same process: with this function, you have a global default graph to which all operations will be added if we don't explicitly create a new graph.

Next, we're ready to add operations to our graph. As it is remembered from working with Keras, we build up our model, and then in compiling it, we define a loss function, an optimizer, and a metric. This now all happens in one step when we work with TensorFlow:

- First, we define placeholders for inputs and labels because we won't put in the "real" data yet. **Remember** that placeholders are values that are unassigned and that will be initialized by the session when we run it. So when we finally run the session, these placeholders will get the values of our dataset that we pass in the run() function!
- Then, we build up the network. We first start by flattening the input with the help of the flatten() function, which will give an array of shape [None, 784] instead of the [None, 28, 28], which is the shape of our grayscale images.
- Activation function :The activation function of a node defines the output given a set of inputs. We need an activation function to allow the network to learn non-linear pattern. A common activation function is a Relu, Rectified linear unit. The function gives a zero for all negative values.

- After we have flattened the input, we construct a fully connected layer that generates logits of size [None, 62]. Logits is the function operates on the unscaled output of previous layers, and that uses the relative scale to understand the units is linear.
- With the multi-layer perceptron built out we can define the loss function. Loss function - after we have defined the hidden layers and the activation function, we need to specify the loss function and the optimizer. The loss function is a measure of the model's performance. The choice for a loss function depends on the task that we have at hand: in this case, you make use of

  sparse_softmax_cross_entropy_with_logits()

- This computes sparse softmax cross entropy between logits and labels. In other words, it measures the probability error in discrete classification tasks in which the classes are mutually exclusive. This means that each entry is in exactly one class. Here, a traffic sign can only have one single label. **Remember** that, while regression is used to predict continuous values, classification is used to predict discrete values or classes of data points. We wrap this function with reduce_mean(), which computes the mean of elements across dimensions of a tensor.
- The optimizer will help improve the weights of the network in order to decrease the loss. Some of the most popular optimization algorithms used are the Stochastic Gradient Descent , ADAM and RMSprop. Depending on whichever algorithm we choose, we'll need to initialize certain parameters, such as learning rate or momentum. In this case, we pick the ADAM optimizer, for which we define the learning rate at 0.001.
- Lastly, we initialize the operations to execute before going over to the training.

```
# Import `tensorflow`
import tensorflow as tf

# Initialize placeholders
x = tf.placeholder(dtype = tf.float32, shape = [None, 28, 28])
y = tf.placeholder(dtype = tf.int32, shape = [None])

# Flatten the input data
images_flat = tf.contrib.layers.flatten(x)

# Fully connected layer & apply relu activation function as tf.nn.relu
logits = tf.contrib.layers.fully_connected(images_flat, 62, tf.nn.relu)

# Define a loss function
loss =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels =
y,

logits = logits))
# Define an optimizer
train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(loss)
```

```
# Convert logits to label indexes
correct_pred = tf.argmax(logits, 1)

# Define an accuracy metric
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

- We have now successfully created our first neural network with TensorFlow!

Now that we have built up our model layer by layer, it's time to actually run it! To do this, we first need to initialize a session with the help of Session() to which we can pass our graph that we defined in the previous section. Next, we can run the session with run(), to which we pass the initialized operations in the form of the init variable that we also defined.

Next, we can use this initialized session to start epochs or training loops. In this case, we pick 201 because we want to be able to register the last loss_value; In the loop, we run the session with the training optimizer and the loss (or accuracy) metric that we defined in the previous section. We also pass a feed_dict argument, with which we feed data to the model. After every 10 epochs, we'll get a log that gives us more insights into the loss or cost of the model.

As we have seen on the TensorFlow basics, there is no need to close the session manually; this is done for us.

We have now successfully trained our model.

We're not entirely there yet; We still need to evaluate our neural network. In this case, we can already try to get a glimpse of how well our model performs by picking 10 random images and by comparing the predicted labels with the real labels.

We can first print them out, but using matplotlib to plot the traffic signs themselves and to make a visual comparison.

However, only looking at random images don't give us many insights into how well our model actually performs. That's why we'll load in the test data and Run predictions against the full test set. Finally calculate the accuracy.

CONCLUSION

Autonomous driving is extremely complex and poses challenging problems,

Core software modules running on Autonomous Vehicles are parallel in nature and run simultaneously.

Autonomous vehicles are modelled on artificial neural networks which have a phenomena of self – organizing system.

The two approaches for deep learning in self – driving cars has been discussed. Our approach is Semantic Abstraction where the problem of Autonomous Driving is broken down into several components.

Finally, Image Classification in Traffic Signs Dataset using Deep Neural Network with TensorFlow has been discussed.

REFERENCES

1. https://github.com/
2. https://github.com/datacamp/datacamp-community-tutorials/.....TensorFlow Tutorial For Beginners.ipynb
3. Gustav Von Zitzewitz : “Survey of neural networks in Autonomous Driving”.

   URL- https://www.researchgate.net/publication/

4. Ivan Dynov : “Is Deep Learing Really the solution for Everything in Self – Driving cars” ? URL – https://www.automotive-iq.com/autonomous-drive/