✨ Member-only story

DEEPPICAR SERIES

DeepPiCar — Part 5: Autonomous Lane Navigation via Deep Learning

Use Nvidia's End-to-End Deep Learning approach to teach our PiCar to navigate lanes autonomously.

David Tian · Follow
Published in Towards Data Science
13 min read · May 3, 2019

Listen Share

Executive Summary

Welcome back! If you have read through DeepPiCar Part 4, you should have a self-driving car that can navigate itself pretty smoothly within a lane. In this article, we will use a deep-learning approach to teach our PiCar to do the same, turning it into a **DeepPiCar**. This is analogous to how you and I learned to drive, by observing how good drivers (such as our parents or driving school coaches) drive and then start to drive by ourselves and learn from our own mistakes along the way. Note that for most of this article, you don't need to have the DeepPiCar to follow along, as we will do the deep-learning on Google's Colab, which is free.

Introduction

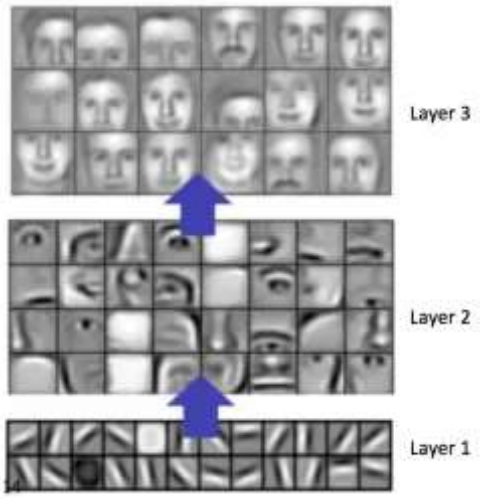
Recall in Part 4, we hand engineered all the steps required to navigate the car, i.e. color isolation, edge detection, line segment detection, steering angle computation, and steering stabilization. Moreover, there were quite a few parameters to hand tune, such as upper and lower bounds of the color blue, many parameters to detect line segments via Hough Transform, and max steering deviation during stabilization, etc. If we didn't tune all these parameters correctly, our car wouldn't run smoothly. Moreover, every time we had new road conditions, we would have to think of new detection algorithms and program them into the car, which is very time consuming and hard to maintain. In the era of AI and machine learning, shouldn't we just able to "show" the machines what to do and have it learn from us, instead of we "tell" it exactly the steps to do it? Luckily, the researchers at Nvidia have demonstrated in this excellent paper that by "showing" a full-scale car how to drive, and the car would learn to drive by itself. This sounds pretty magical, right? Let's see how this was done, and how to apply it to our DeepPiCar.

The Nvidia Model

At the high level, the inputs to the Nvidia model are video images from DashCams onboard the car, and outputs are the steering angle of the car. The model uses the video images, exacts information from them, and tries to predict the car's steering angles. This is known as a **supervised machine learning program**, where video images (called **features**) and steering angles (called **labels**) are used in training. Because the steering angles are numerical values, this is a **regression** problem,

instead of a **classification** problem, where the model needs to predict if a dog or a cat, or which one type of flower is the in the image.

At the core of the NVidia model, there is a **Convolutional Neural Network** (CNN, **not** the cable network). CNNs are used prevalently in image recognition deep learning models. The intuition is that CNN is especially good at extracting visual features from images from its various layers (aka. filters). For example, for a facial recognition CNN model, the earlier layers would extract basic features, such as line and edges, middle layers would extract more advanced features, such as eyes, noses, ears, lips, etc, and later layers would extract part or all of a face, as illustrated below. For a comprehensive discussion on CNN, please check out Convolutional Neural Network's Wikipedia page.



The CNN layers used in the Nvidia model is very similar as above, as it extracts lines and edges in its early layers and complex shapes in its later layers. The fully connected layers function can be thought of as a controller for steering.

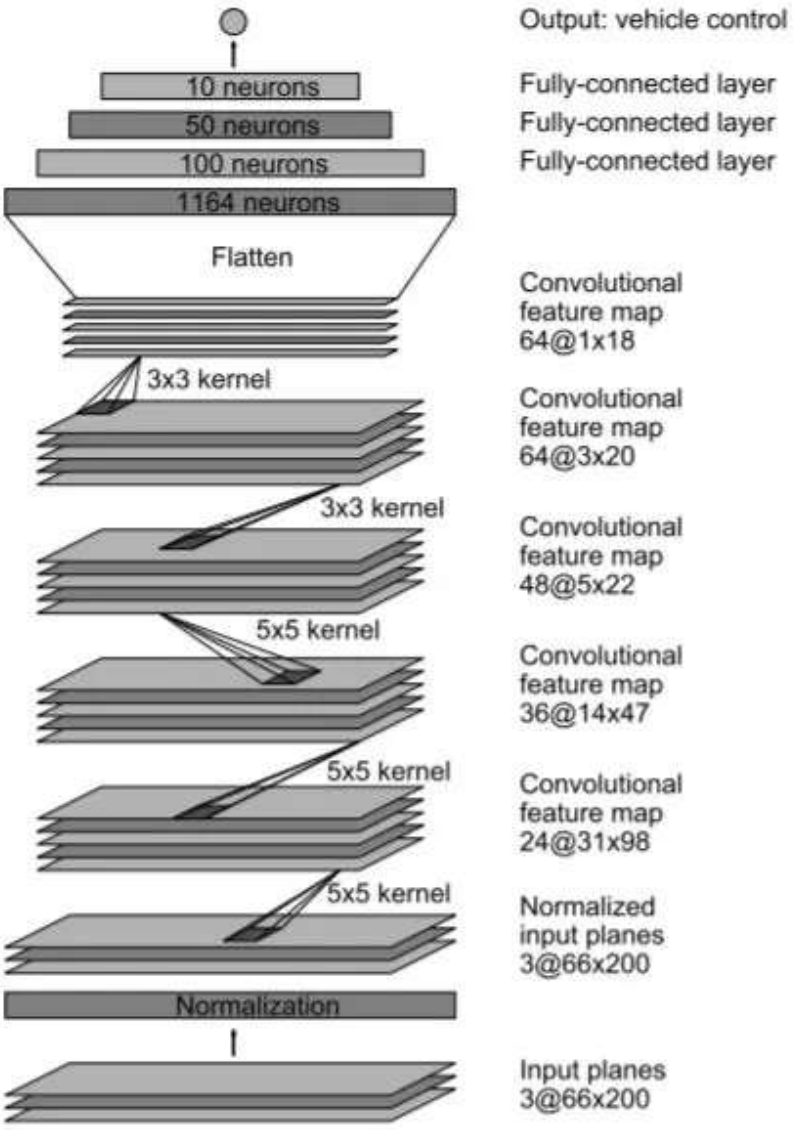
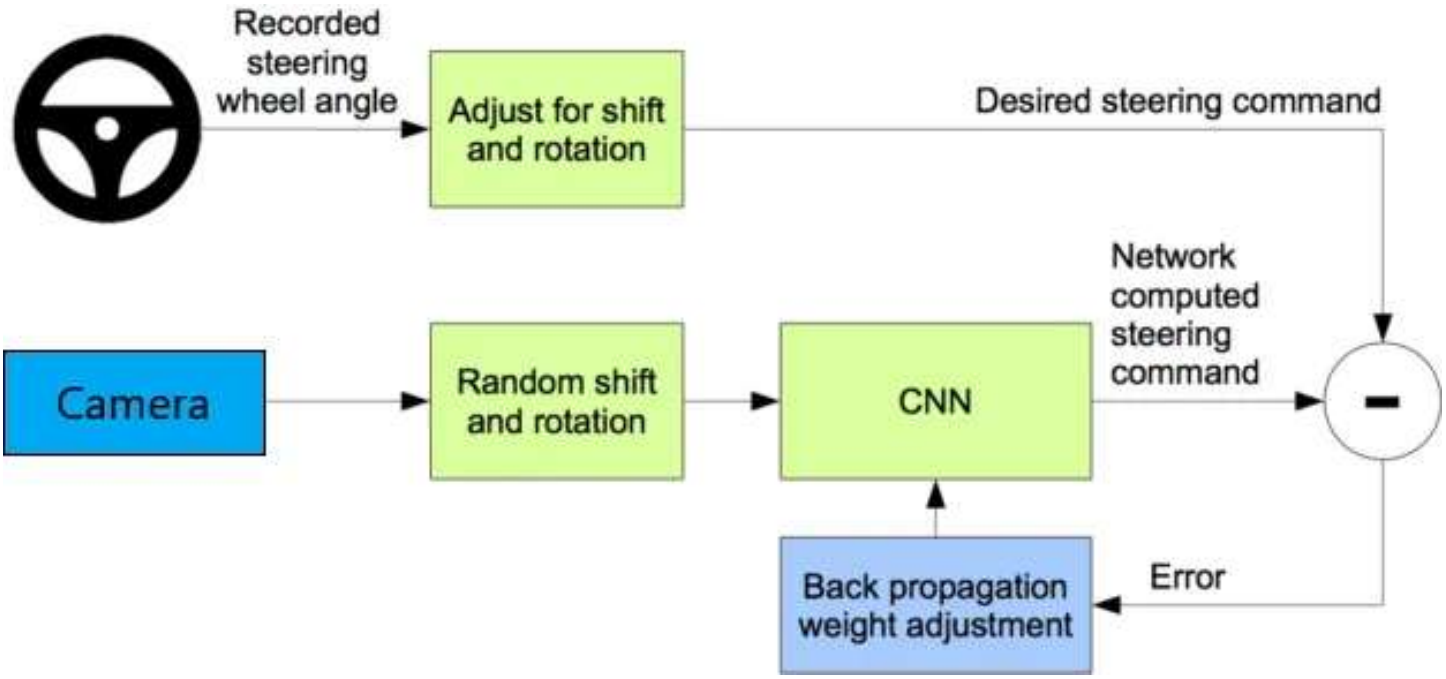


Figure 4: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

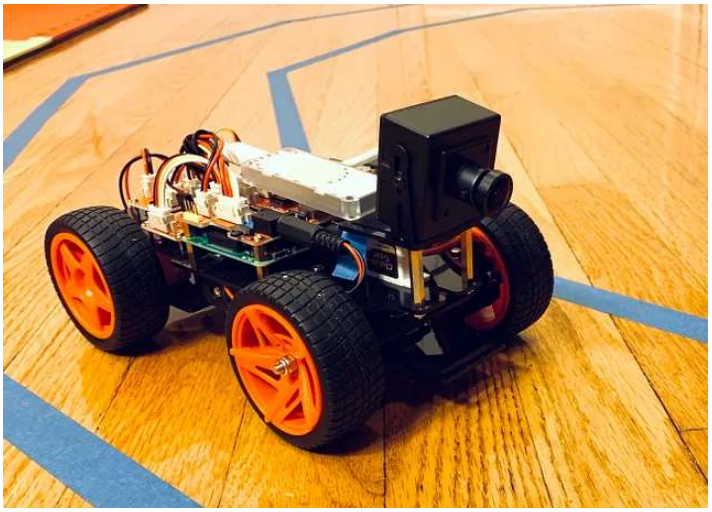
The above diagram is from Nvidia's paper. It contains about 30 layers in total, not a very deep model by today's standards. The input image to the model (bottom of the diagram) is a 66x200 pixel image, which is a pretty low-resolution image. The image is first normalized, then passed through 5 groups of convolutional layers, finally passed through 4 fully connected neural layers and arrived at a single output, which is the model predicted steering angle of the car.



This model predicted angle is then compared with the desired steering angle given the video image, the error is fed back into the CNN training process via **backpropagation**. As seen from the diagram above, this process is repeated in a loop until the errors (aka **loss or Mean Squared Error**) is low enough, meaning the model has learned how to steer reasonably well. Indeed, this is a pretty typical image recognition training process, except the predicted output is a numerical value (**regression**) instead of the type of an object (**classification**).

Adapting the Nvidia Model for DeepPiCar

Other than in size, our DeepPiCar is very similar to the car that Nvidia uses, in that it has a dash cam, and it can be controlled by specifying a steering angle. Nvidia collected its inputs by having its drivers drove a combined 70 hours of highway miles, in various states and multiple cars. So we need to collect some video footage of our DeepPiCar and record the correct steering angle for each video image.

See the similarities? No Hands on the steering wheel!

Data Acquisition

There are multiple ways to collect training data.

One way is to write a remote control program so that we can remotely steer the PiCar, and have it save down the video frame as well as the car's steering angles at each frame. This is probably the best way since it would be simulating a real person's driving behavior, but requires us to write a remote control program. An easier way is to leverage what we have built in Part 4, which is the lane follower via OpenCV. Since it runs reasonably well, we can use that implementation as our "model" driver. Machine learning from another **machine**! All we have to do is to run our OpenCV implementation on the track a few times, save down the video files and the corresponding steering angles. We can then use them to train our Nvidia model. In Part 4, `deep_pi_car.py` will automatically save down a video file (AVI file) every time you run the car.

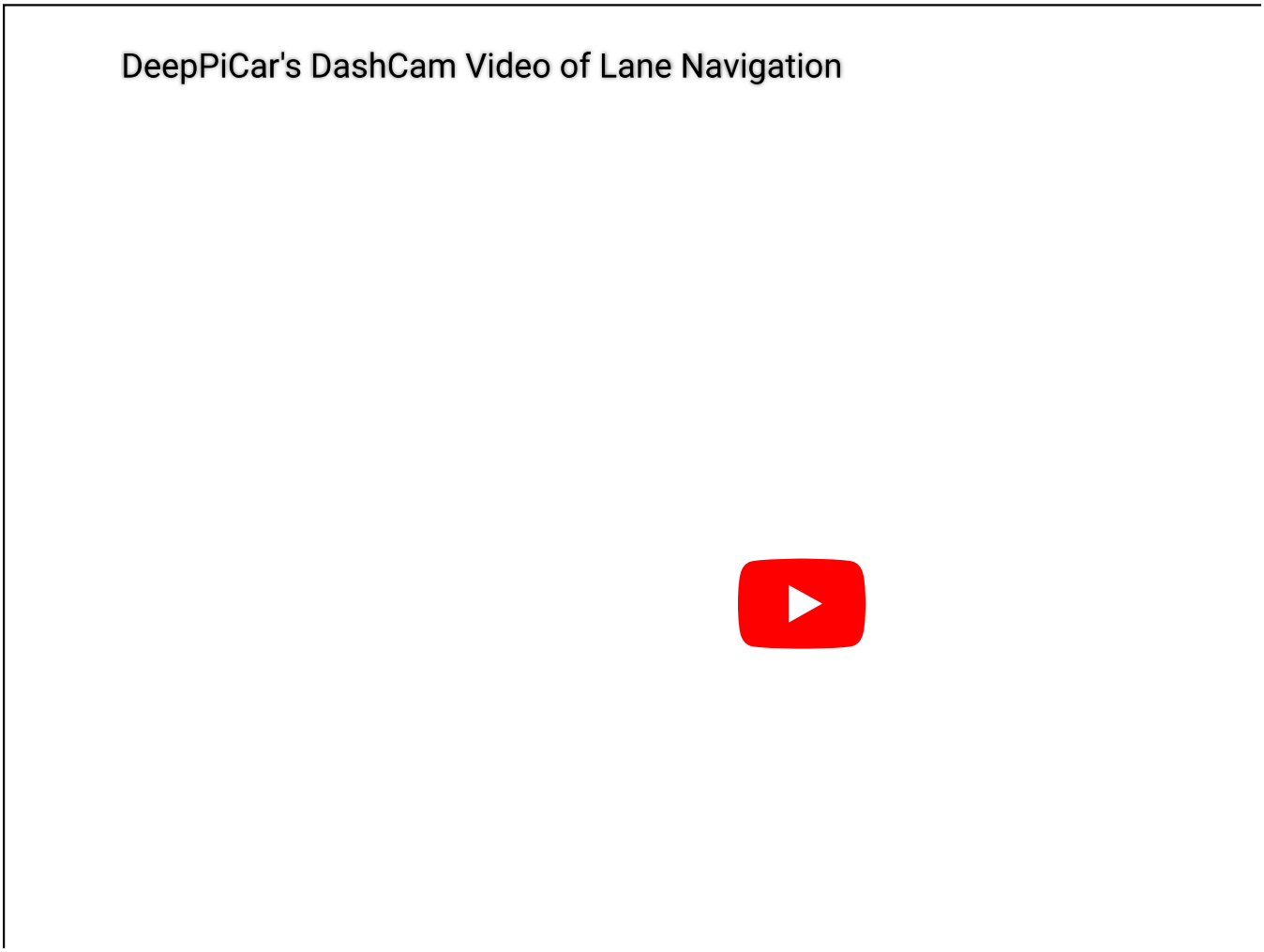
Here is the code to take a video file and save down the individual video frames for training. For simplicity, I embed the steering angle as part of the image file name, so I don't have to maintain a mapping file between image names and steering angles.

```python
import cv2
import sys
from hand_coded_lane_follower import HandCodedLaneFollower

def save_image_and_steering_angle(video_file):
    lane_follower = HandCodedLaneFollower()
    cap = cv2.VideoCapture(video_file + '.avi')

    try:
        i = 0
        while cap.isOpened():
            _, frame = cap.read()
            lane_follower.follow_lane(frame)
            cv2.imwrite("%s_%03d_%03d.png" % (video_file, i, lane_follower.curr_steering_angle)
            i += 1
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
    finally:
        cap.release()
        cv2.destroyAllWindows()

if __name__ == '__main__':
    save_image_and_steering_angle(sys.argv[1])
```

save_training_data.py hosted with ❤ by GitHub view raw

Assuming you have a recorded dashcam video called video01.avi from Part 4,

Here is a sample DashCam video.



Here is the command to save still images from it and tag it with the steering angle.

```
pi@raspberrypi:~/DeepPiCar/driver/code $ python3
save_training_data.py
```


suffix are the steering angle. From below, we can tell the car was turning left, as angles are all smaller than 90 degrees, which is confirmed watching the DashCam video from above.

```
pi@raspberrypi:~/DeepPiCar/driver/code $ ls
~/DeepPiCar/models/lane_navigation/data/images |more
video01_000_085.png
video01_001_080.png
video01_002_077.png
video01_003_075.png
video01_004_072.png
```

```
video01_005_073.png
video01_006_069.png
```

Training/Deep Learning

Now that we have the features (video images) and labels (steering angles), it is time to do some deep learning! In fact, this is the first time in this DeepPiCar blog series that we are doing deep learning. Even though **Deep Learning** is all the hype these days, it is important to note **it is just a small part of the whole engineering project**. Most of the time/work is actually spend on hardware engineering, software engineering, data gathering/cleaning, and finally wire up the predictions of the deep learning models to production systems (like a running car), etc.

To do deep learning model training, we can't use the Raspberry Pi's CPU, and we need some GPU muscle! Yet, we are on a shoestring budget, so we don't want to pay for an expensive machine with the latest GPU, or rent GPU time from the cloud. Luckily, Google offers some of GPU and even TPU power **for FREE** on this site called Google Colab! Kudos to Google for giving us machine learning enthusiasts a great playground to learn!

Colab is a free cloud-based Jupyter Notebooks that let you write and train deep learning models in Python. The popular python libraries supported are TensorFlow, Keras, OpenCV, and Pandas, etc. The cool thing is that TensorFlow for GPU is already preinstalled, so you don't have to spend hours to mess with pip or the CUDA driver or software set up, and can just jump right in to train your models.

Without further ado, let's start the Jupyter Notebook. I am assuming that the readers are relative well versed in Python and Keras library.

Here is my entire end-to-end deep learning lane navigation notebook on GitHub. I will cover the key parts of it below.

Import Libraries

Firstly, we need to import python libraries that we will use during the training process.

Load Training Data

Then we need to load the training data. For this article, I have uploaded a sample set of the generated image files to my GitHub, so that readers can clone it and follow along. Remember, the image files are named as `videoXX_FFF_SSS.png`, where `videoXX` is the name of the video, `FFF` is the frame number in the video, and `SSS` is the steering angle in degree. The resultant training data is read in `image_paths` and

`steering_angles` variables. For example, `video01_054_110.png` means that this picture came from `video01.avi` video file, it is the 54th frame, and the steering angle is 110 degree (turning right).

Split into Train/Test Set

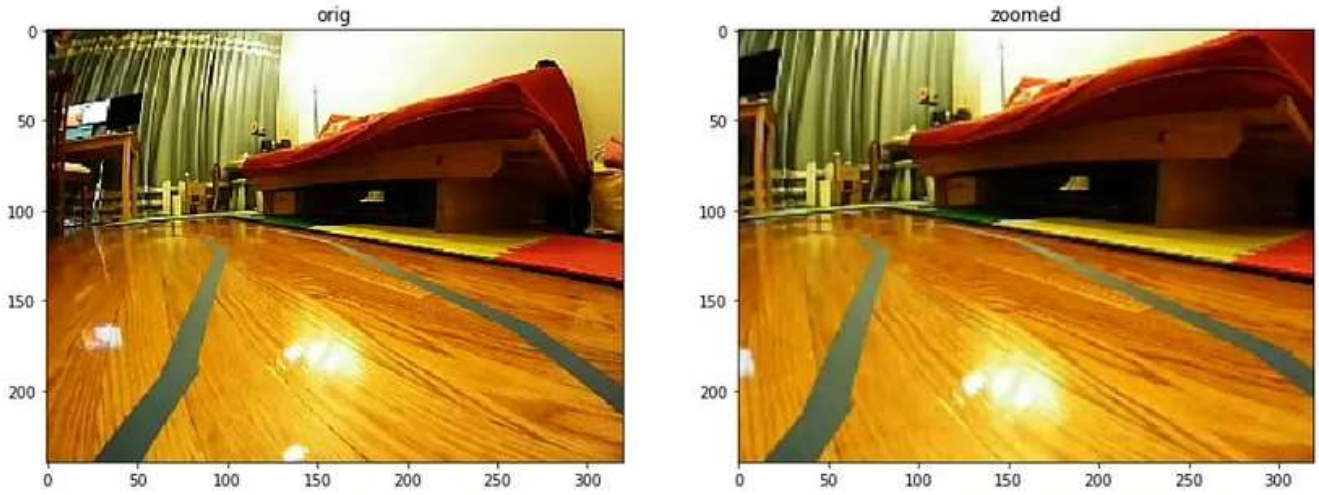
We will split the training data into training/validation sets with a 80/20 split with sklearn's `train_test_split` method.

Image Augmentation

The sample training data set only has about 200 images. Clearly, that's not enough to train our deep learning model. However, we can employ a simple technique, called Image Augmentation. Some of the common augmentation operations are zooming, panning, changing exposure values, blurring, and imaging flipping. By randomly applying any or all of these 5 operations on the original images, we can generate a lot more training data from our original 200 images, which makes our final trained model much more robust. I will just illustrate zoom and flip below. Other operations are quite similar and covered in my Jupyter notebook in GitHub.

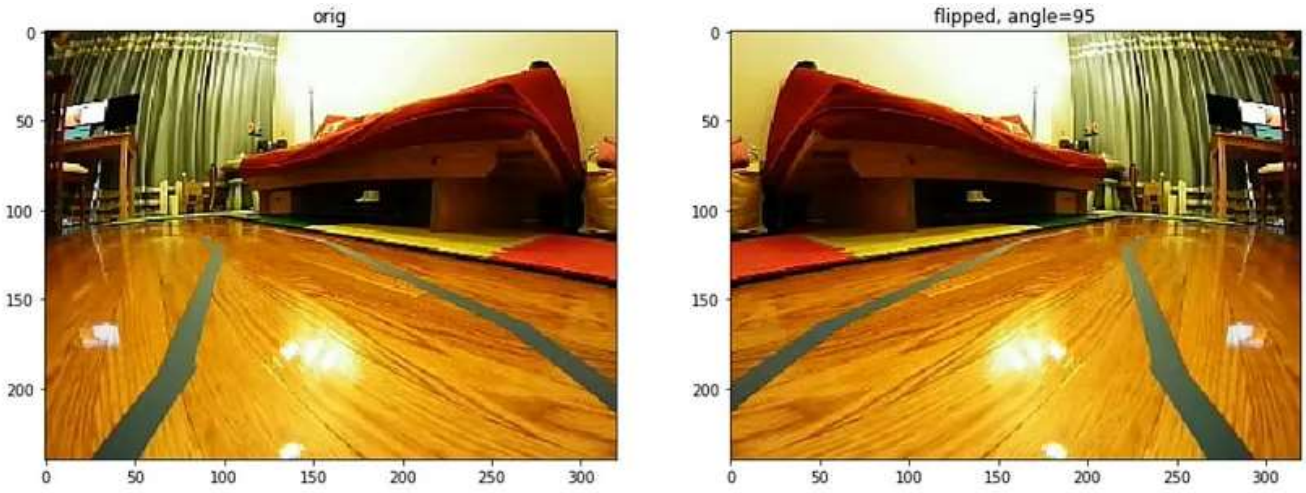
Zoom

Here is the code for the random zoom, between 100% and 130%, and the resultant zoomed image (right).



Flip

Here is the code for the random flip. Note that flip operation is different from other image augmentation operations because when we flip the image, we need to change the steering angle. For example, the original image below (left) has a lane line curving left, hence a steering angle of 85. But when we flip the image, the lane line is now pointing right, so the correct angle is `180 -original_angle`, which is 95 degrees.



We have a function to combine all the augmentation operations together, so an image can have any or all operations applied to it.

Image Preprocess

We also need to change our images into the color space and size that the Nvidia model accepts. First, the Nvidia research paper calls for input images in 200x66 pixel resolution. Similar to what we did in Part 4, the top half of the image is not relevant to predicting the steering angle, so we will just crop it out. Secondly, it calls for the images to be in YUV color space. We will simply use `cv2.cvtColor()` to do that. Lastly, it requires us to normalize the images.

Nvidia Model

Here we present the Nvidia Model Architecture again, so we can easily compare it with our model in code.

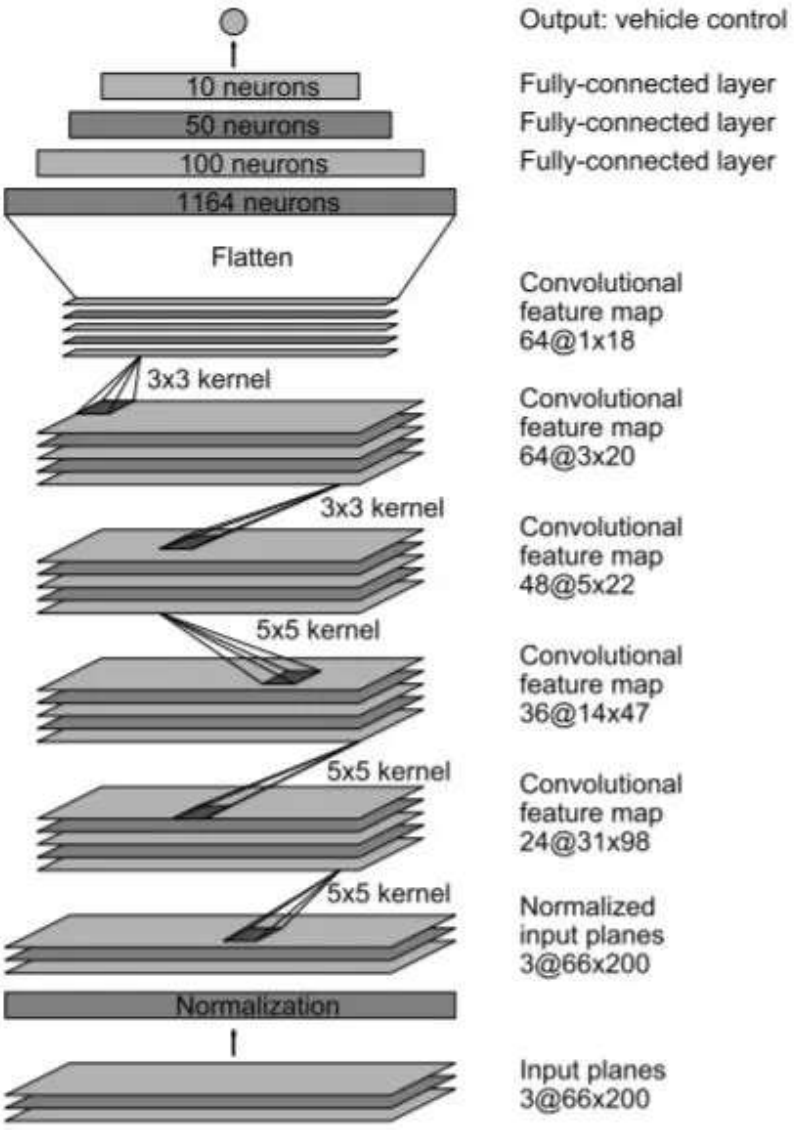
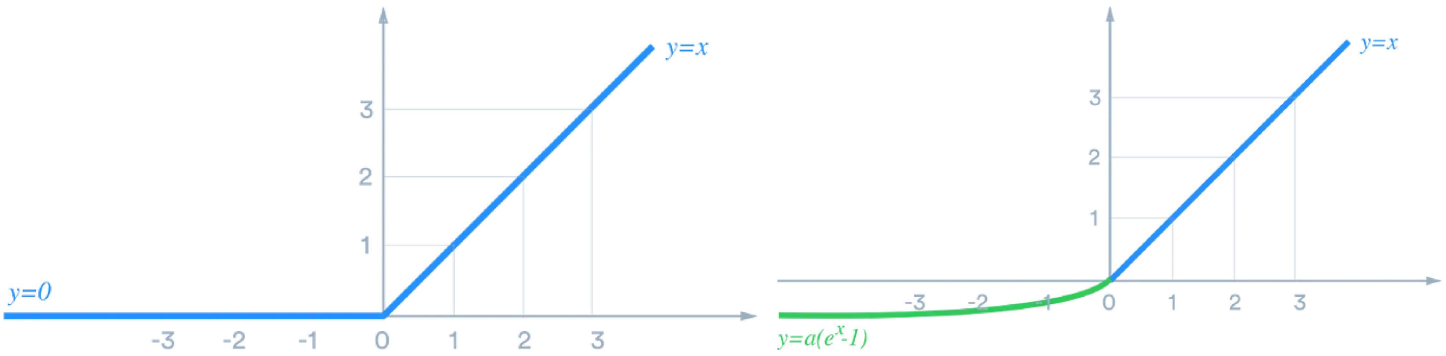


Figure 4: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

Note that we have fairly faithfully implemented the Nvidia model architecture, except that we removed the normalization layers, as we would do that outside of the model, added a few dropout layers, to make the model more robust. The loss function we use is Mean Squared Error (MSE) because we are doing regression training. We also used the ELU (Exponential Linear Unit) activation function instead of the familiar ReLU (Rectified Linear Unit) activation function because ELU doesn't have the "dying RELU" problem when x is negative.



ReLU (left) and ELU (right)

When we create the model and print out its parameters list, it shows that it contains about 250,000 parameters. This is a good check that each layer of our model is created as we expect.

```
Model: "Nvidia_Model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 31, 98, 24)        1824
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 14, 47, 36)        21636
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 5, 22, 48)         43248
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
dropout (Dropout)            (None, 3, 20, 64)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
flatten (Flatten)            (None, 1152)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 10)                510
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 11
=================================================================
Total params: 252,219 Trainable params: 252,219 Non-trainable
params: 0
```

Training

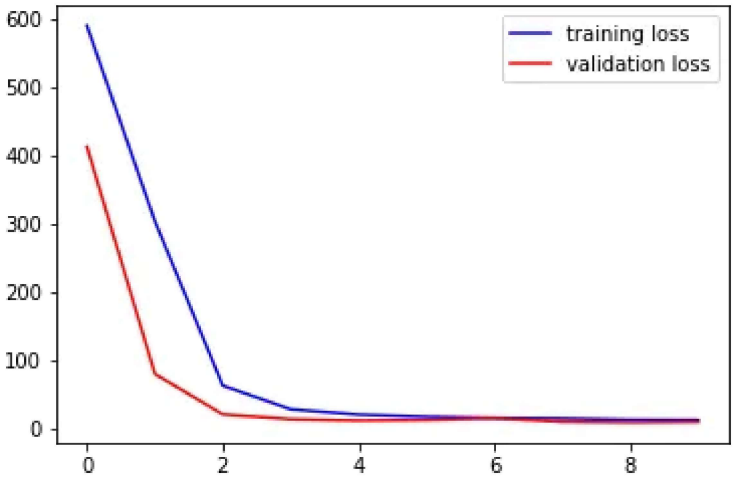
Now that both the data and model are ready, we will start to train the data.

For those who have used Keras to train deep-learning models, we usually use the `model.fit()` command. But notice that today we used `model.fit_generator()` command. This is because we are not using a static set of training data, our training data is generated dynamically from our original 200 images via image augmentation discussed earlier. For this to work, we need to create a helper function that does the augmentation and then return a new batch of training data to `model.fit_generator()` on each iteration.

Here is the code of this helper, `image_data_generator()`.

Evaluate the Trained Model

After training for 30 min, the model will finish the 10 epochs. Now it is time to see how well the training went. First thing is to plot the loss function of both training and validation sets. It is good to see that both training and validation losses declined rapidly together, and then stayed very low after epoch 5. There didn't seem to be any overfitting issue, as validation loss stayed low **with** training loss.



Another metric to see if our model performed well is the R^2 metric. An R^2 close to 100% means the model is performing pretty well. As we can see our model, even

with 200 images has an R^2 of 93% which is pretty good, which is primarily because we used image augmentation.

```
mse       = 9.9
r_squared = 93.54%
```

Here is the full Jupyter Notebook source code on GitHub to train the deep learning lane navigation model, it is best to open it with Google Colab directly. This way, you can run the code yourself once you supply it with your Google credentials.

On the Road

Whether a model is really good needs to be ultimately tested when the rubber meets the road. Below is the core logic to drive the PiCar. Comparing with our hand-coded lane navigation implementation, all the steps (~200 lines of code) of detecting blue color, detecting lane, and computing steering angles, etc are gone. Replacing it are these simple commands, `load_model` and `model.predict()`. Of course, this is because all the calibrations are done at the training stage, and the 3Mb HDFS formatted trained model file contains a whopping 250,000 parameters.

Here is the full source code of lane navigation code using the trained deep learning model. Note there are a few helper and test functions in the file to help with display and testing.

Here is the video of DeepPiCar running in the lane. Yes, it now **truly** deserves the name, **DeepPiCar**. We can see it was doing fine most of the way but veered off the lane a bit toward the end. Because this is using deep learning, to fix this issue, all we have to do is to give the model more video footage of good driving behavior, either by driving the car via remote control or by using a better hand-coded lane navigator. If we want the car to navigate on a white/yellow marked road, then we can just give it video feeds with white/yellow marked roads and steering angles, and the model will learn that as well without us having to hand tune color mask for both white and yellow colors. How cool is that!

What's Next

In this article, we taught our DeepPiCar to autonomously navigate within a lane simply by letting it "observe" how a good driver drives. Contrasting the deep learning lane navigation implementation with the hand-tuned implementation from our previous article, it is much much shorter and simpler to understand. Its accuracy is also quite high, achieving 94% R^2. Of course, we had to train the deep learning model first, which contained 250,000 parameters. But it learned all the parameters simply by observing how we drive. In the next article, we will build another important feature of an autonomous car, which is to observe and respond to its surroundings, most commonly, respond to traffic signs and pedestrians. Hope to see you in Part 6!

Here are the links to the whole guide:

Part 1: Overview

Part 2: Raspberry Pi Setup and PiCar Assembly

Part 3: Make PiCar See and Think

Part 4: Autonomous Lane Navigation via OpenCV

Part 5: Autonomous Lane Navigation via Deep Learning (This article)

Part 6: Traffic Sign and Pedestrian Detection and Handling

Self Driving Cars | Deep Learning | Deep Pi Car | Raspberry Pi | Machine Learning

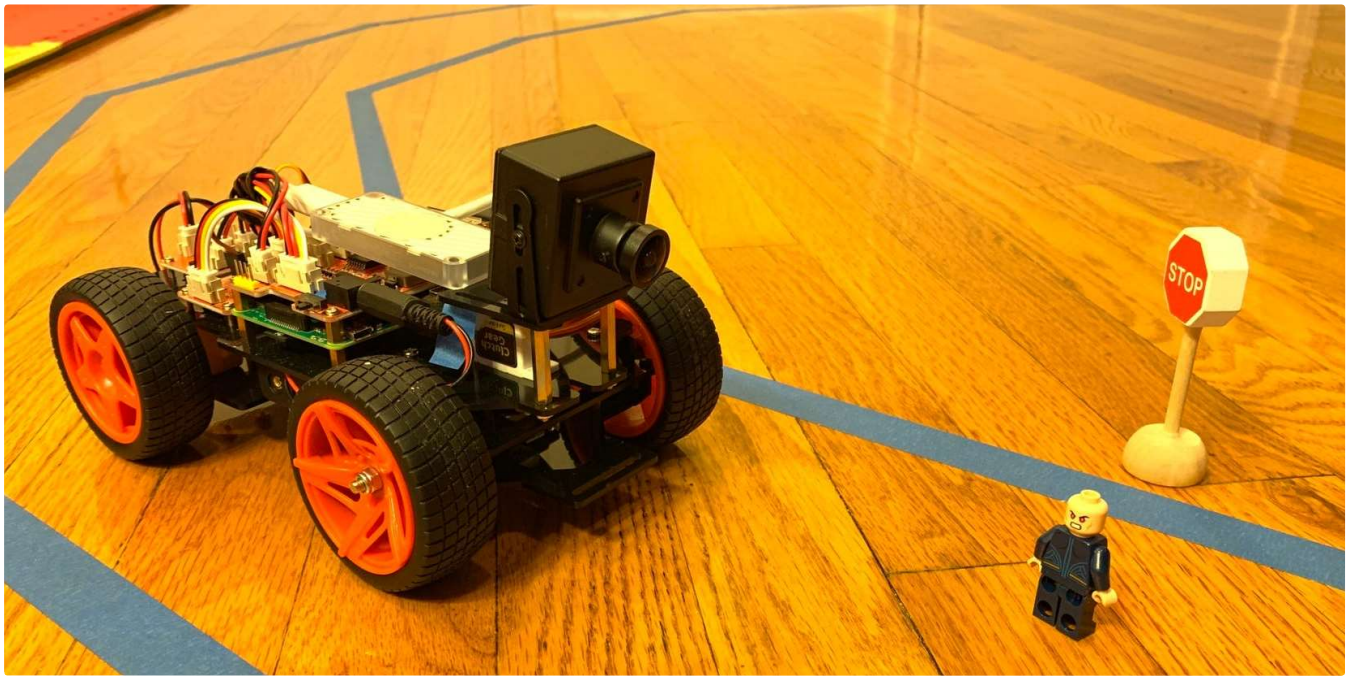
Follow

Written by David Tian

656 Followers · Writer for Towards Data Science

Hacker, tinkerer, and engineer. I am passionate about machine learning, AI, and anything technology related.
DeepPiCar GitHub: https://github.com/dctian

More from David Tian and Towards Data Science



David Tian in Towards Data Science

DeepPiCar — Part 1: How to Build a Deep Learning, Self Driving Robotic Car on a Shoestring Budget

An overview of how to build a Raspberry Pi and TensorFlow powered self driving robotic car

6 min read · Apr 19, 2019


1.3K


5

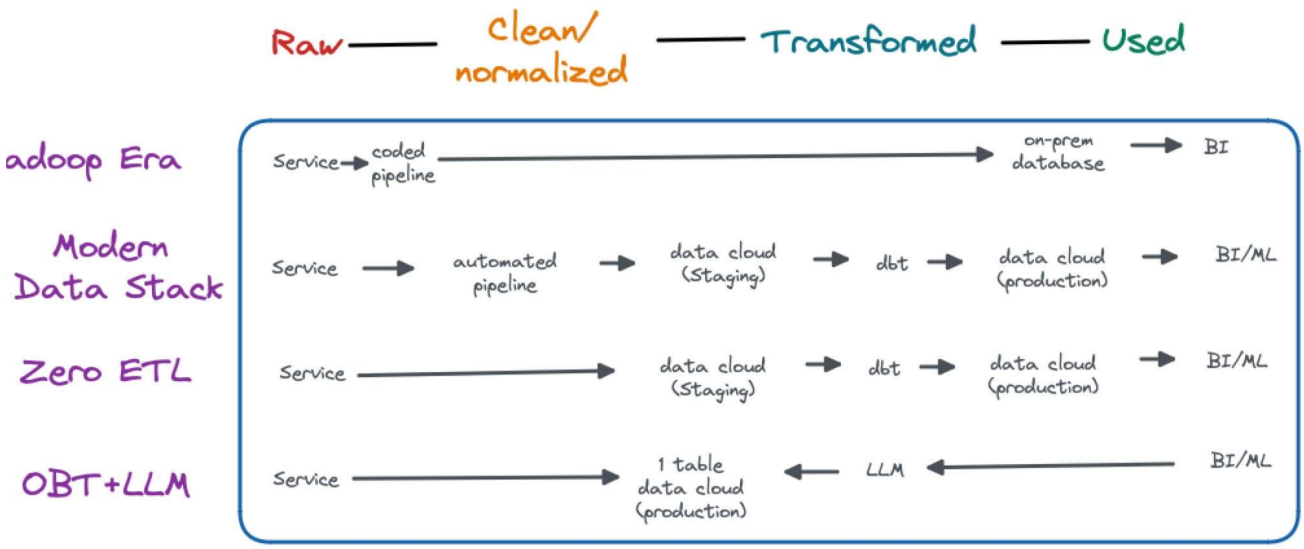
Jacob Marks, Ph.D. in Towards Data Science

How I Turned My Company's Docs into a Searchable Database with OpenAI

And how you can do the same with your docs

15 min read · Apr 25

2K 26



Barr Moses in Towards Data Science

Zero-ETL, ChatGPT, And The Future of Data Engineering

The post-modern data stack is coming. Are we ready?

9 min read · Apr 4

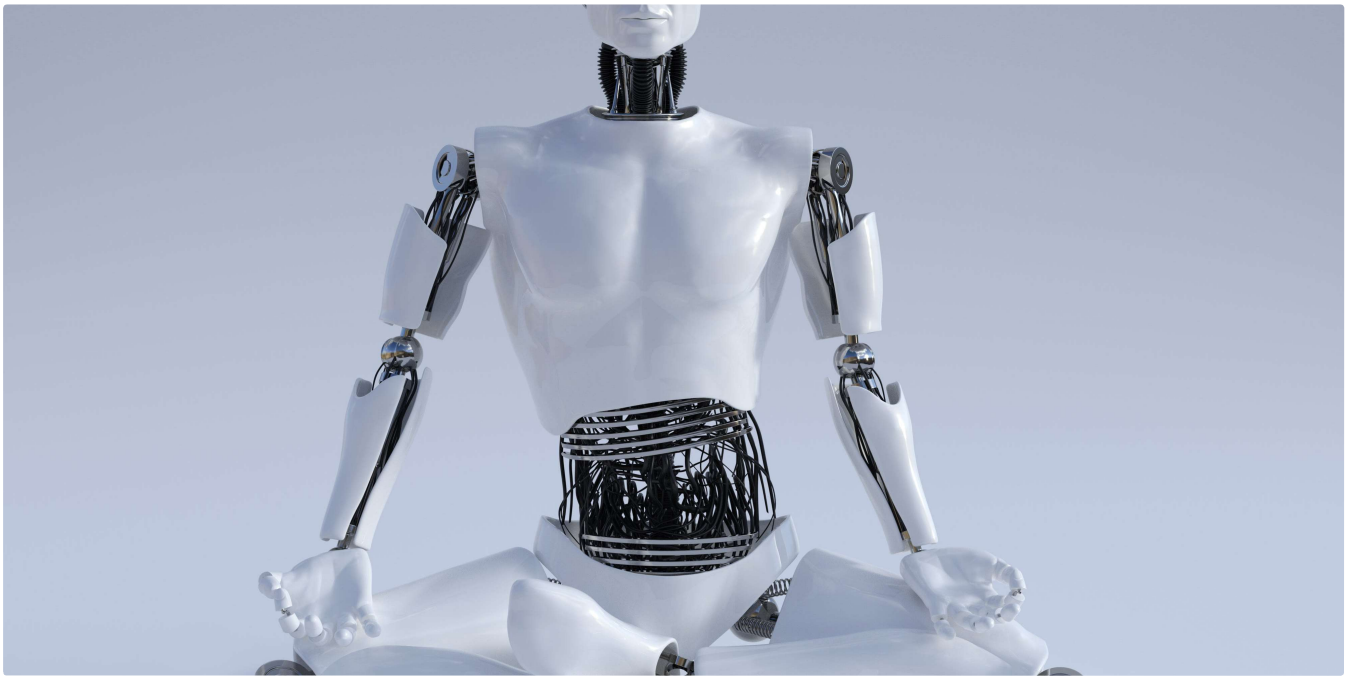
1.1K 22

David Tian in Towards Data Science

DeepPiCar — Part 4: Autonomous Lane Navigation via OpenCV

Use OpenCV to detect color, edges and lines segments. Then compute steering angles, so that PiCar can navigate itself within a lane.

· 15 min read · May 3, 2019



The PyCoach in Artificial Corner

You're Using ChatGPT Wrong! Here's How to Be Ahead of 99% of ChatGPT Users

Master ChatGPT by learning prompt engineering.

✨ · 7 min read · Mar 17

16.8K 298

Alexander Nguyen in Level Up Coding

Why I Keep Failing Candidates During Google Interviews...

They don't meet the bar.

4 min read · Apr 13

3.2K 101

Unbecoming

10 Seconds That Ended My 20 Year Marriage

It's August in Northern Virginia, hot and humid. I still haven't showered from my morning trail run. I'm wearing my stay-at-home mom...

4 min read · Feb 16, 2022

46K 748

Aleid ter Weel in Better Advice

10 Things To Do In The Evening Instead Of Watching Netflix

Device-free habits to increase your productivity and happiness.

✦ · 5 min read · Feb 15, 2022

17.6K 231

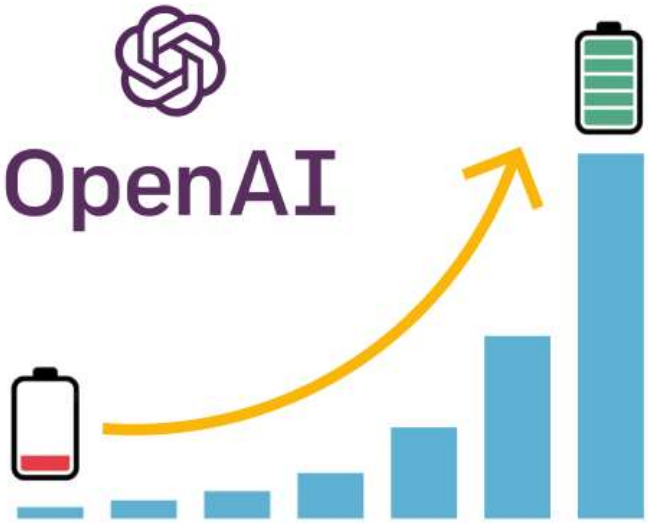
Mark Schaefer

20 Entertaining Uses of ChatGPT You Never Knew Were Possible

Our RISE community has been on fire, exploring the breathtaking possibilities of ChatGPT. The uses of ChatGPT are simply endless and...

✦ · 12 min read · Dec 12, 2022

20K 370



 Josep Ferrer in Geek Culture

5 ChatGPT features to boost your daily work

And how to enhance your code quality using it

✦ · 8 min read · Jan 17

4.3K 66

See more recommendations