



IT-Project

Autonomously driving Remote Control Car

LÖHR Tim, BOHNSTEDT Timo, PALPANES Ioannis

Abstract—Im Rahmen unseres IT-Projekts beschäftigen wir uns mit der Entwicklung eines autonom fahrendes ferngesteuertes Auto. ToDo

Index Terms—Machine Learning, IT-Project, RC-Car, RaspberryPi, Autonomously driving

1. PROJECT BACKGROUND AND MOTIVATION

ToDo

2. LITERATURE SURVEY

2.1 Reinforcement Learning

ToDo

3. HARDWARE

3.1 RC Car

3.1.a Technical Details: ToDo

3.1.b Servomotor: ToDo

3.2 Raspberry Pi

Even though by now the Raspberry Pi 4 is out, we still use the Raspberry Pi 3B+, because we are sure that it is compatible with our other microcontrollers and software.

We use a 16GB SanDisc to store the operating system and the machine learning model on the Raspberry Pi.

3.2.a Technical Details: The Raspberry Pi 3B+ is equipped with a lot of different extensions. It has a Wi-Fi module to communicate with the Python webserver and a fairly fast processor to run the neural network and classify where to drive next. The Processor is the Broadcom BCM2837B0, Cortex-A53 with 1.4GHz and it has 1GB LPDDR2 SDRAM.

Furthermore for the connectivity the Pi has extended 40-pin GPIO header. The weight is only 42 gramms, which make it easy to attach it onto the RC car.

3.2.b Cameramodule: ToDo

3.3 Servodriver

We use the SunFounder PCA9685. The price for this microcontroller is approximately 13 euro on the most famous marketplaces. The servodriver is needed to produce the PWM (Pulse width modulation) signals for our Raspberry Pi. It acts as joint for the RC cars's servomotor and the Raspberry Pi.

3.3.a Technical Details: The SunFounder PCA9685 consists of many different Pins [1]. It is powered by the GND and the VCC pin. The VCC is uses voltage of 3-5V. Furthermore are three control pins on the board, namely the:

- SCL - I2C clock pin
- SDA - I2C data pin
- OE - Output enable

3.4 Cable Management

3.4.a Servomotor to Servodriver: A real model car has always a servomotor as motor. This servomotor receives input from two pins and each of the two are powered separately with V+ and the GND pin. So in total the servomotor has six pins. Whereas the servodriver has 3x16 rows of pins available on it. Each are the three output pins:

- V+ - Power supply
- GND - Ground
- PWM

We connect the servomotor pins accordingly onto two of the rows of the servodriver. This six pins in total control the entire behavior of the car. One of the three-pin-blocks is responsible for the steering and the other one is responsible for the speed of the car.

3.4.b Servodriver to Raspberry Pi: There are four pins that we need to connect from the Raspberry Pi onto the servodriver. Two out of those four pins are just for the power supply. We connect the 5V pin from the raspberry Pi to the VCC pins, which accordingly powers the SunFounder servodriver. To make a power supply work we always need the ground pin (GND) as well. The other two pins communicate the data from Raspberry Pi to servodriver and vice versa. The communication pins work with the I2C bussystem, which is

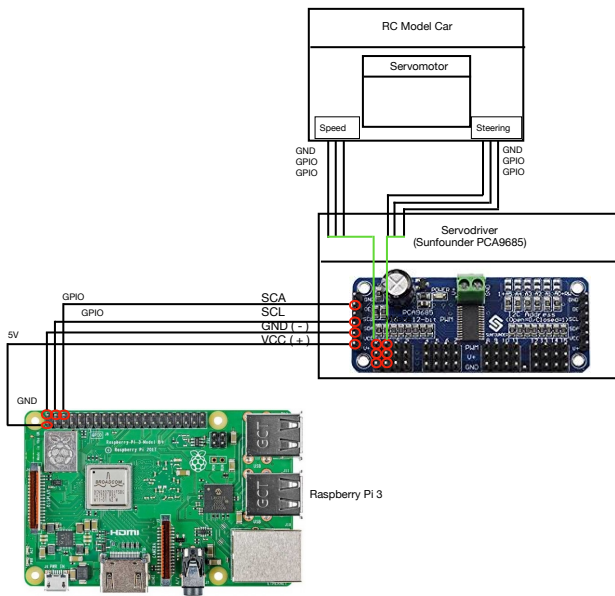


Fig. 1. Cable Management

used by the SDL (serial data) pin and the SCL (serial clock). The SDL 10Bit addressing bus sends the data and the SCL sets up the clocking time respectively. Both the Raspberry Pi and the servodriver have the SDL and the SCL connector pins, so we can easily plug them in.

4. SOFTWARE ARCHITECTURE

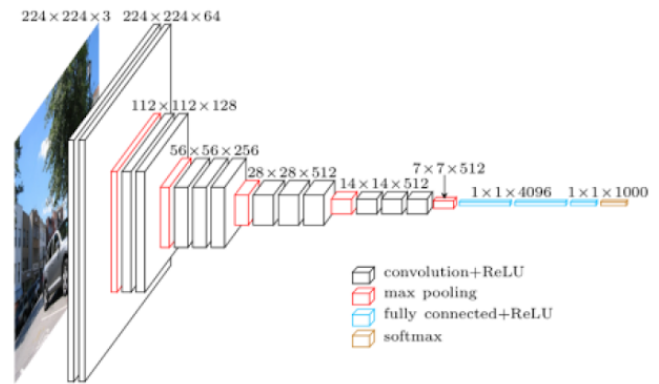
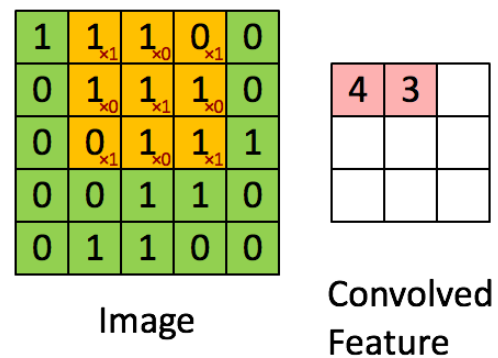
4.1 Python Files

4.2 Webserver

4.3 Unity

5. DEEP LEARNING

To solve the image classification task, we are using a convolutional neuronal network (CNN). CNN's are a particular case of feed-forward neural networks [2]. On a very fundamental level, we can say that a feed-forward neural network - as the most machine learning models - is a function [3]. The function of a feed-forward neural network can simply be described as $y = f(x, \phi)$. CNN's and feed-forward neural networks are estimating parameter values. So for $y = f(x, \phi)$ we estimate the parameter ϕ [2]. Through those estimated parameter values we are receiving a function back with the smallest possible difference between the predicted output values and the defined output values. A function that measures the difference between the expected - and the defined output is a so-called loss function [2]. The problem description gives already a mathematical definition (??) from the accuracy loss function, which is used by us. CNN's and classical feed-forward neuronal networks differ in their basis of calculation. Feed-forward neural networks are using matrix multiplication, whereas CNN's are using convolutions. Convolutional layers, pooling layers, and fully-connected layers are specific layers for a CNN [4].

Fig. 2. VGG16 Model Architecture from <https://arxiv.org/pdf/1409.1556.pdf>Fig. 3. Feature Map derived from <http://deeplearning.stanford.edu>

5.0.a Model Architecture: We decided to use the VGG16 model architecture [5] (Figure 2).

In 2014, an implementation of the architecture won the ILSVR(Imagenet) competition. As we could already see in the global leaderboard that there are quite impressive results, we still decided to use this classic example. The complexity is rather small in comparison to other designs which are provided in the ranking. Note that - as mentioned in the "Problem Description", the goal of our project was not to get an outstanding result of the accuracy. The main goal was to understand the principals behind deep learning and to get a feeling of what the state of the art (in image recognition) is. So, because the VGG16 [5] is known as the best visualizable model architecture, we thought it would be the best choice to accomplish our goal in this way. The model architecture is also known for getting rid of a vast amount of hyper-parameters. Instead, it is focusing on tiny filter size and a small strate. The technical details "Problem Description", of the different layers are providing more details about its implementation.

5.0.b Convolutional Layers: The convolutional Layer performs the extracting of features from the input matrix into a feature map. For this procedure, we use matrix multiplication in the form of the dot product and a filter (feature detector, kernel) [2].

In picture 3 can be seen a calculation example of how

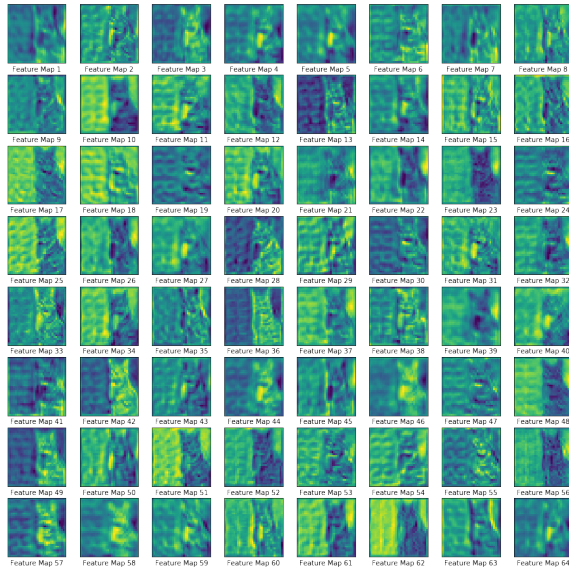


Fig. 4. Feature Map from CNN Layer 1

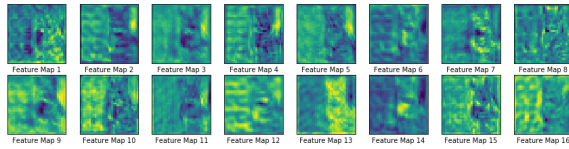


Fig. 5. Feature Map from CNN Layer 2

convolution is applied to a matrix. We are iterating with the filter over the matrix, calculating the scalar product and writing the result into a feature map. An activation function like the rectified linear unit (ReLU) normalizes the feature map after we performed the convolution. Normalization guarantees a feature map which is not a linear transformation of its input value. If this is the case, we only have a linear problem which can be easily solved, but with a wrong results. In other words, the input values get just multiplied by coefficients. Note that there are plenty of reasons for activation functions in a neuronal network. We are giving a closer look at this topic at our section "Activation Functions". In Figure 4 you can see how our pictures had changed after applying the first convolutional layer, before and after doing normalization.

This represents the 64 first feature maps of a random picture from the dataset. We can observe that the filters already focus on certain points on the cat's body. We further used this feature map as input for the second CNN layer. The output of the second feature map (figure 5) further sharpens the filters. For example we can see that feature map 2 until 5 focuses on the cat itself, whereas map 8 puts its focus on the cats ears.

5.0.c Pooling Layers: It is common to add a pooling layer in-between the convolutional layers periodically. The function exists to avoid overfitting and step by step reduce the size of the input image by reducing the number of parameters

and computations of the network [6]. We use three pooling layers with filters of size 3x3 applied with a stride of 0 and a padding set to *same*. This downsamples every depth slice in the input by 2 along both width and height, discarding 50% of the activations.

5.0.d Fully-Connected Layers: The Dense layer is the last layer of our model. It is also called the *fully connected layer*. Neurons in the fully connected layer have full connections to all activations in the previous layer, as in the normal neural networks. Their activation functions will then be computed by a matrix multiplication which is followed by a bias offset. For the ten different animals we respectively have ten fully connected neurons in the dense layer.

5.0.e Kernel Regularization: There are many different regularizations to prevent the neural network from overfitting. For our project we chose the *L2 regularization* because it is the most common form. It can be integrated by penalizing the squared magnitude of all parameters directly in the objective. So, for every weight w in the network, the term $\frac{1}{2}\lambda w^2$ gets added to the objective, where λ is the regularization strength. It's usual to see $\frac{1}{2}$ in the front, because the gradient of this term with respect to w is simply λw instead of $2\lambda w$. The L2 penalizes the peaky weight vectors and prefers the diffuse weight vectors [7]. During the gradient descent weight update, the L2 regularization has the meaning, that every weight is decayed linearly ($w += -\text{lambda} * w$) towards zero.

5.0.f Batch Normalization: A technique developed by Ioffe and Szegedy [7] is called Batch Normalization properly initializes neural networks by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. Normalization is a simple differentiable operation. It allows to use higher learning rates at the beginning and is less vulnerable to a bad initialization. In other words, neural networks that implement batch normalization layers are significantly more robust. Additionally, batch normalization can be interpreted as a preprocessing step on every layer of the network. Batch normalization yields in general, a substantial improvement in the training process.

5.0.g Activation Functions: There a couple of widely used activation functions like tanh, sigmoid function, ReLU or the ELU. For our model we decided to use the *ReLU* activation function. The Rectified Linear Unit (ReLU) has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero. There are plenty of pro's and con's for the ReLU:

- (+) Compared to tanh/sigmoid neurons that need to compute expensive operations like the exponentials, the ReLU can be implemented by directly thresholding a matrix of activations at zero.
- (+) It was found to notably accelerate the convergence of stochastic gradient descent (SGD) compared to the tanh/sigmoid functions. That is, because of its linear, non-saturating form.

- (-) Sadly, ReLU units can be weak during training and possibly “die”. For example, a large gradient streaming through a ReLU neuron could cause the weights to update in a dead end, so that it will never activate again. If this happens, then the gradient streaming through the unit will forever be zero. The ReLU units can irreversibly die during training since they can get eliminated off the data manifold. With a good scheduling setting of the learning rate this is less likely to happen.

5.0.h Loss function: Our compiled Keras model uses the *cross-entropy-loss* [8].

$$L_i = f_{y_i} + \log \sum_j e^{f_j} \quad (1)$$

where we are using the notation f_j to mean the j -th element of the vector of class scores f . The full loss for the dataset is the mean of f_i over all training examples together with a regularization term $R(W)$. The cross-entropy loss, or also called log loss, measures the performance of our classification model with the output as probability values between zero and one. The cross-entropy loss increases as the predicted probability diverges from real value labels.

5.0.i Optimizer: There are many possible optimizer suitable for our task. The common ones are the RMSprop, Adam or the stochastic gradient descent (SGD) [9]. We decided to use the SGD to minimize the loss by computing the gradient with respect to a randomly selected batch from the training set. This method is more efficient than computing the gradient with respect to the whole training set before each update is performed.

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_i) & \text{if } i = j \\ -p_j p_i & \text{if } i \neq j \end{cases} \quad (2)$$

$$\frac{\partial L}{\partial o_i} = p_i - y_{oh_i} \quad (3)$$

For the derivation of the cross entropy loss, y_{oh} is the one-hot encoded representation of the class labels.

5.0.j Softmax: The softmax normalizes the class probabilities to one and it has a probabilistic interpretation.

$$f_j(z) = \frac{e^{z_i}}{\sum_k e^{z_k}} \quad (4)$$

The exponential values can very quickly explode to an infinite large number, for example e_{1000} . To fix this issue, it takes a one-dimensional vector of arbitrary length (in z) and puts it into a vector of values between zero and one that sum together to one. The cross-entropy loss that includes the softmax function, hence to minimize the cross-entropy-loss between the estimated class probabilities.

5.0.k Flatten: In between the convolutional layer (CNN) and the fully connected layer (Dense), there is a *Flatten layer*. The Flattening layer transforms a two-dimensional matrix of features into a one-dimensional vector that can be respectively streamed into the fully connected neural network

classifier, which are our ten fully connected animal neurons.

6. AUTONOMOUSLY DRIVING

7. EVALUATION

8. CONCLUSION

ACKNOWLEDGMENT

The authors would like to thank Prof Dr. Florian Gallwitz from the University of Applied Science - Georg Simon OHM in Nuremberg for a really good supervising of our group.

REFERENCES

- [1] [Online]. Available: <https://cdn-learn.adafruit.com/downloads/pdf/16-channel-pwm-servo-driver.pdf>
- [2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [3] V. D. Visin and Francesco, “A guide to convolution arithmetic for deep learning,” 2018. [Online]. Available: <https://arxiv.org/pdf/1603.07285.pdf>
- [4] Y. LeCun, L. Bottou, G. B. Orr, and K. R. Müller, *Efficient BackProp*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. [Online]. Available: https://doi.org/10.1007/3-540-49430-8_2
- [5] K. S. . A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” 2015. [Online]. Available: <https://arxiv.org/pdf/1409.1556.pdf>
- [6] H. W. Gu and Xiaodong, “Max-pooling dropout for regularization of convolutional neural networks.” [Online]. Available: <https://arxiv.org/pdf/1512.01400.pdf>
- [7] S. I. Szegedy and Christian, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” 2015. [Online]. Available: <https://arxiv.org/pdf/1502.03167.pdf>
- [8] Z. Z. Sabuncu and M. R., “Generalized cross entropy loss for training deep neural networks with noisy labels.” [Online]. Available: <https://papers.nips.cc/paper/8094-generalized-cross-entropy-loss-for-training-deep-neural-networks-with-noisy-labels.pdf>
- [9] M. R. Gorti and S. Krishna, “Faster gradient descent via an adaptive learning rate,” 2017. [Online]. Available: <http://www.cs.toronto.edu/~mravox/p4.pdf>