TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

IT-Projekt
Autonmously 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, Autonomusly driving**

1. Project Background and Motivation

ToDo

2. Literature Survey

2.1 Reinforcement Learning

3. Hardware

3.1 RC Car

3.1.a Technischer Aufbau:
3.1.b Servomotor:

3.2 Raspberry Pi

3.2.a Technical Details:
3.2.b Cameramodule:

3.3 Servodriver

3.3.a Technical Details:

3.4 Cable Management

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 [**?**]. On a very fundamental level, we can say that a feed-forward neural network - as the most machine learning models - is a function [**?**]. The function of a feed-forward neural network can simply be described as $y = f(x, \emptyset)$. CNN's and feed-forward neural networks are

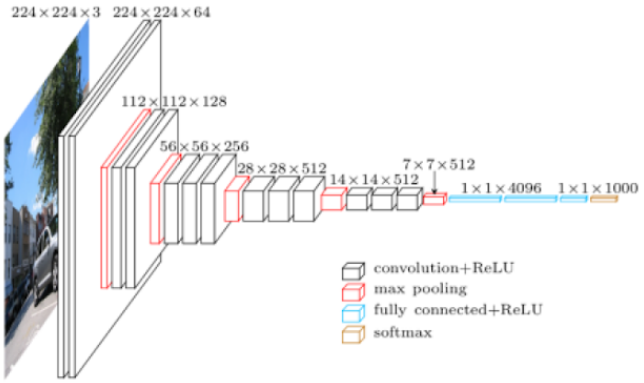

Fig. 1. VGG16 Model Architecture from https://arxiv.org/pdf/1409.1556.pdf

estimating parameter values. So for $y = f(x, \emptyset)$ we estimate the parameter $\emptyset$ [**?**]. 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 [**?**]. To compare our model accuracy with other model, we are using the following evaluation criteria:

$$Acc(f, D) = \frac{1}{m} \sum_{m}^{i=1} \mathbb{I} \left[y^{(i)} = f(x^{()i)}) \right] \quad (1)$$

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 [**?**].

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

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

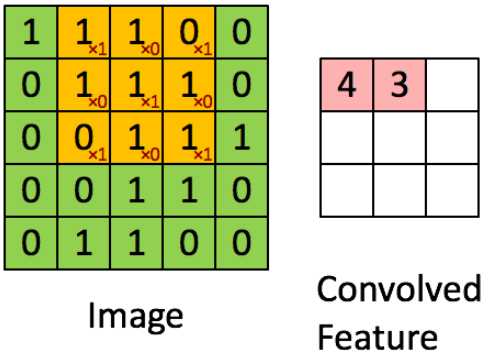

Fig. 2. Feature Map derived from http://deeplearning.stanford.edu

provided in the ranking. Note that - as mentioned in the "Problem Desciption", 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 [**?**] 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 Desciption", 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) [**?**].

In picture **??** can be seen a calculation example of how 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 **??** 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 **??**) further sharpens the filters. For example we can see that feature map 2 until 5 focuses

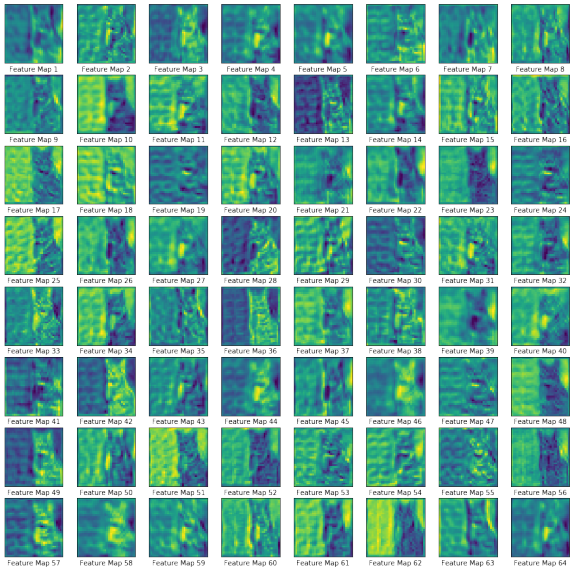

Fig. 3. Feature Map from CNN Layer 1

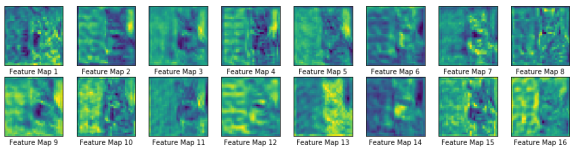

Fig. 4. Feature Map from CNN Layer 2

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 [**?**]. 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 [**?**]. During the gradient descent weight update, the L2 regularization has the meaning, that every weight is decayed linearly (*w += -lambda * w*) towards zero.

5.0.f Batch Normalization: A technique developed by Ioffe and Szegedy [**?**] 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* [**?**].

$$L_i = f_{y_i} + log \sum_j e^{f_j} \tag{2}$$

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 RMSprob, Adam or the stochastic gradient descent (SGD) [**?**]. 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.

$$\frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1-p_i) & if \quad i = j \\ -p_j p_i & if \quad i \neq j \end{cases} \tag{3}$$

$$\frac{\partial L}{\partial o_i} = p_i - y\_oh_i \tag{4}$$

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.

$$f_j(z) = \frac{e^{z_i}}{\sum_k e^{z_k}} \tag{5}$$

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 probabilites.

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.