End-to-end Driving Controls Predictions from Images

Jan Felix Heyse and Maxime Bouton
CS229 Final Project Report

December 17, 2016

Abstract

Autonomous driving is a promising technology to improve transportation in our society. In this project we propose an end-to-end approach to learn how to steer a car autonomously from images. We trained Convolutional Neural Networks (CNNs) to output steering wheel angle commands from front camera images centered on the road. Finally, we explored some smoothing of the predictions by exploiting the temporal structure of the data.

1 Introduction and Previous Work

In 2015, the DOT reported that more than 35,000 people died on US roadways due to crashes. 94% of these crashes happened because of human error [1]. With the progress that has been made over the past years in machine learning and artificial intelligence, we are convinced that autonomous driving has got the potential to significantly reduce that number. One of the biggest challenges of autonomous driving today is to handle the uncertainty when acting in the real world.

In most of the current self-driving technologies, the input provided to the system is based on human engineered features. This refered to as mediated perception [2]. The control of the vehicle is based on criteria such as vehicles position and velocity and position of the ego car in the lane. Based on these heuristic criteria the embedded controller will optimize some cost functions to drive the vehicle safely. However, these cost function and heuristic models are hard to find and since they are based on human heuristic they are no guarantee of optimality. Another drawback is that the resulting systems are not well scalable to different scenarios.

This leads to the development of *end-to-end* approaches ([2] and [3]). The idea consists in removing every modeling assumptions and engineered features and let the system learn how to accomplish the desired task using supervised learning techniques. Using an end-to-end approach could lead to a robust driving capability while relaxing any modeling assumptions. The work described in [3] is a promising step in this direction, using Convolutional Neural Networks (CNNs).

The input of the system is a raw image from a camera on the autonomous vehicle, and the output is a driving control. In this project we will focus on predicting steering wheel angle commands from a center camera view. We will train a predictor using an available labeled dataset and analyze its generalization performance.

2 Dataset

2.1 Description

The dataset for our project has been provided by Udacity [4]. As part of their efforts towards an open source self-driving car, Udacity broke down the problem into a series of challenges. The goal of one of these challenges is the prediction of steering angles from images, and the provided datasets contain the data neccessary for our purposes. In particular, they comprise images from three cameras at the front of the car at 20 frames per second with a resolution of 640×480 pixels, as well as the steering wheel angles and further information relevant for car controls at a higher rate. From the three front cameras, positioned at the left, center, and right of the car, we decided to use the center camera. Due to computational limitations we also decided to use one of the datasets only, which was 12:40 minutes of driving on a sunny day.

Figure 1: Block diagram representing the input output behavior of our system

2.2 Pre-processing

After extracting the data from the available dataset we used three pre-processing steps to prepare it for our calculations. First, we interpolated the steering angles to the time stamps of the front camera images. Next, we created clean folders with the used data only, i.e. the center camera images and their associated steering angles, and renamed the files to make them more easily processable. Lastly, we went through the chosen dataset and excluded those sections in which the car stopped at red traffic lights and where the car turned onto a different road. The first was to avoid a bias on these situations due to a very high number of practically identical images and steering angles. The second was because intentional turns of the driver were not part of this project. In order to predict them further information on the intention of the driver would be necessary. The final dataset we used consisted of 12,200 samples corresponding to 10:10 minutes of driving.

Besides this general pre-processing, the different approaches required further individual processing which will be presented in the respective sections.

3 A Classification Task

The task of predicting the driving controls was approached in three different ways of increasing complexity. The first two treated the task as a ternary classification problem with the three classes *left*, *straight*, and *right*, while the third treated the task as a regression, aiming at the exact prediction of the particular steering command. The motivation for starting with a classification problem was that it

would be simpler and therefore more appropriate to find a baseline approach and that it would provide the prediction accuracy as an intuitively understandable metric.

In order to obtain the classification labels from the continuous steering command variable, we set the thresholds based on some image-steering command pairs and on the steering command distribution in general. Table 1 shows the distribution of the labels among the used samples.

	Left	Straight	Right	Total
Number	3753	5619	2828	12200
Fraction	30.8%	46.1%	23.2%	100%

Table 1: Classification label distribution

3.1 SVM as a Baseline

Our first approach to the image classification was using a Support Vector Machine (SVM). We converted the images from color to grayscale and divided every image into sub-images of size 20×20 pixels. As features we reported then minimum, maxmimum, and average grayscale values of the sub-images. The dataset was split up into a training and a test set (see table 2).

	Training	Test
Number	9760	2440
Fraction	80%	20%

Table 2: SVM training and test set sizes

The SVM learning was implemented using the scikit-learn machine learning python library. We chose a Gaussian Kernel with a γ value of 0.0001.

3.2 A CNN based Approach

	Training	Validation	Test
Number	7320	1464	3416
Fraction	60%	12%	28%

Table 3: CNN training, validation, and test set sizes

CNNs are a model similar to regular Neural Networks, which does particularly well on images. Just like Neural Networks, Cnns use hidden layers between the input and the output. One key difference is that in case of regular Neural Networks these hidden layers are all fully connected. When working with images as input, this input has the three dimensions image height, image width, and the color representation (in our case the tree BGR values). Fully connecting all layers then would lead to a hardly manageable number of connections. CNNs instead connect each neuron in a hidden layer for the most part only to a small part of all the neurons in the preceding layers, depending on the type of layer. The types we used for our problem were the following [5]:

- *Convolutional Layers* (`CONV`) comprise a specified number of different filters, each moving through the input and acting at each position on a small, local subset of the input only. Further parameters are the filter sizes as well as the stride when sliding through the input.
- *Rectified Linear Unit Layers* (`ReLU`) apply the activation function $x \mapsto \max\{0,x\}$.
- *Parametric Rectified Linear Unit Layers* (`PReLU`) extend the `ReLU` layers with a small negative slope at the otherwise zero-slope parts. This new slope is learned.
- *Pooling Layers* (`POOL`) reduce the spatial size of their input and thereby the complexity of the whole model, which helps preventing overfitting. The reduction factors are the parameters to this type of layer.
- *Dropout Layers* (`DROP`) help to prevent overfitting by setting neurons to zero with a specified probability.
- *Fully Connected Layers* (`FC`), as the name indicates, have neurons connected to all neurons of the preceding layer. They are typically used at the very end to compute the class scores, and it is necessary to specify the output dimension.

A common layer pattern is the following [5]:

```
INPUT->[[CONV->RELU]*N->POOL?]*M
   ->[FC->RELU]*K->FC->OUTPUT
```

Here `N`, `M`, `K` are integers indicating the number of repetitions of the pattern.

The optimizer:
In order to train the network we used categorical cross entropy as the objective function. This is a multiclass version of the logarithmic logistic loss that is computed the following way:

$$CrossEntropy(y,p) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij}\log p_{ij}$$

In this equation m is the number of samples, n is the number of classes, y_{ij} is 1 if example i was assigned to label j, and p_ij is the probability of assigning i to j. We then used Stochastic Gradient Descend (SGD) on this loss function with learning rate decay after each update. We also decided to use the momentum update version as recommended in [5] and [6] for faster convergence.

4 A Regression Task

Using CNNS for the classification task gave us some first experience and a good insight into the work with CNNs in general. However, for autonomous driving in the real world it is insufficient to make a prediction to the three used classes only. We therefore decided to switch to a regression task, which consists in predicting the actual steering command instead of just classifying the direction. The output space is now continuous. As the input still has the same shape it made sense to start with a network architecture similar to that for the classification task. To find the best architecture for the regression we gradually increased the complexity of the network based on the work of [3] and [5].

The main modifications with the network used for regression are the following:

- Change of the activation layers: The ReLU activation layers have a zero gradient zone that can lead to neurons dying off during the optimization process. That is why for the CNN regression we switched to PReLU layers. These address that issue by adding a very small slope at the otherwise

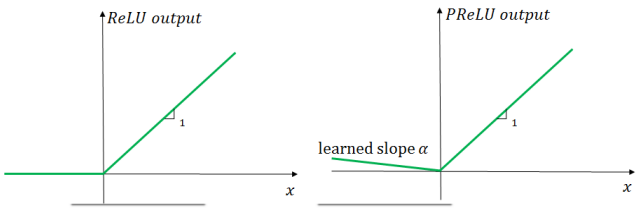

Figure 2: Activation functions of the ReLU and PReLU layers

zero-slope zones as represented in figure 2. The value of the slope is learned during the optimization.

- Change of the optimizer: SGD was converging fast enough for the classification task but since the output space is much larger we needed another optimizer with better convergence properties to deal with the regression problem. The optimizer we chose was Adam, following the recommendations in [5]. Experiments showed that Adam outperformed SGD on minimizing the mean squared error loss.

5 Results

5.1 Classification

We started off by orientating at a configuration used for the comparable cifar10 case [6]; the main modification we made was taking out the dropout layers. The architecture used for the CNN classification showed generally good results. After only 30 epochs we obtained 97% accuracy on the test set. The final architecture for the CNN classification problem is the following:

```
INPUT->[[CONV->RELU]*2->POOL]*2
  ->[FC->RELU]*1->FC->OUTPUT
```

Optimizing the categorical cross entropy with SGD lead to the learning history presented in figure 3 and figure 4.

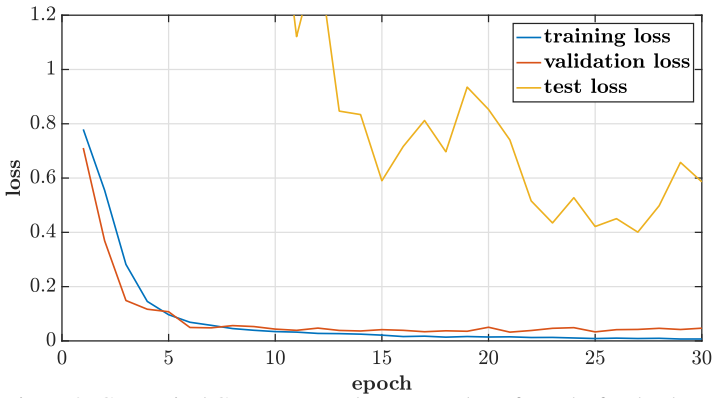

Figure 3: Categorical Cross Entropy loss vs number of epochs for the three parts of the dataset

Figure 6 shows some output of the layers in the CNN. We can see that the network learned that the interesting part of the image (the red parts in the last layer) are in the center. If we look at the input image, this corresponds to the border of the road.

5.2 Regression

After dealing with the classification task we tried to predict the actual steering command *i.e.* having a continuous output space. This task is more complicated and required more iterations in the network architecture and parameters. Naturally, we decided to start

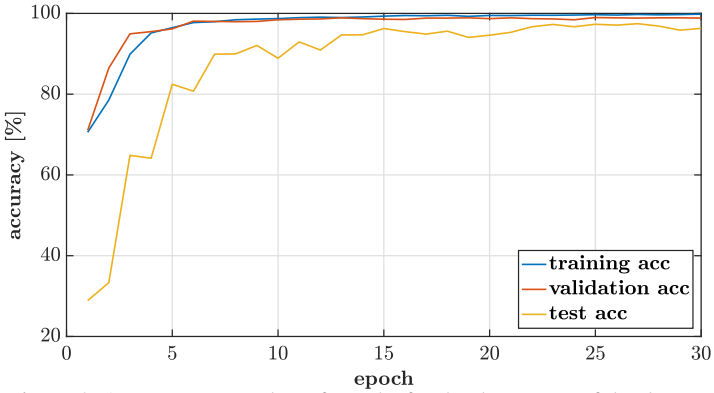

Figure 4: Accuracy vs number of epochs for the three parts of the dataset

with the same architecture that we used for the classification. Unfortunately, this lead to poor performance (Model 1 in table 5), converging very quickly to a high mean squared error (MSE) value. We suspected that the ReLU activation layer was responsible because the rapid convergence could be an indicator of many neurons dying off due to the possible zero-gradient through them. In order to fix this we switched to PReLU activation layers (see section 4). The result was a significant improvement in the loss (for both the training and the test loss).

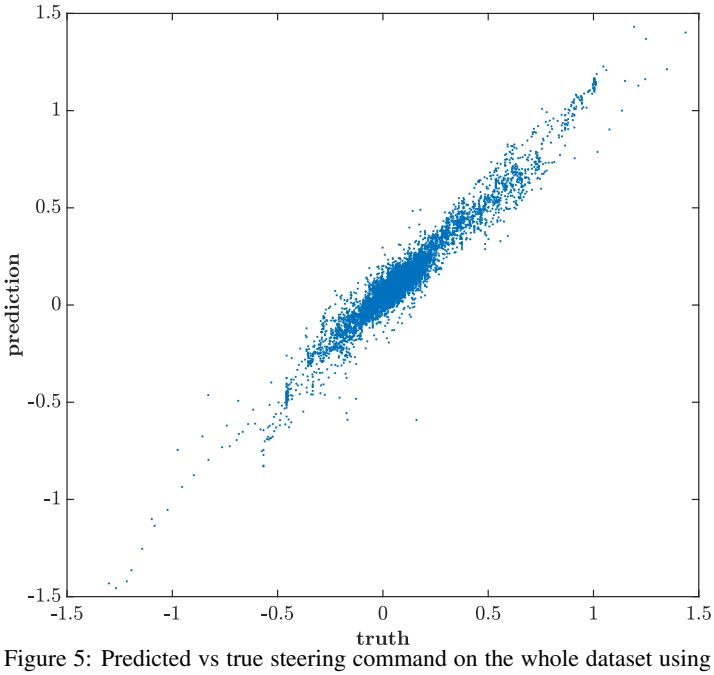

Figure 5: Predicted vs true steering command on the whole dataset using the final regression model

The second step of tuning the network architecture was based on the work from [3] and recommendations from [5]. Basically, we gradually increased the complexity of the network by adding layers and varying parameters. Since the output space is much larger, the intuition was that we need to find more hidden feature in order to reach the desired target. The presented architectures differ in the number of `[CONV->RELU]*2->POOL` blocks, the convolutional filter size, and the use of dropout layers. The architectures of the different iterations are summed up in table 4. Table 5 presents the corresponding performances, with the model 5 architecture performing the best. The figures 5, 7, and 8 visualize the results and the learning history for this architecture.

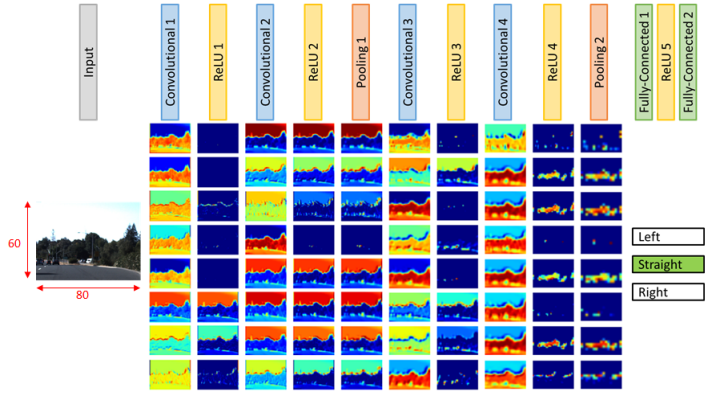

Figure 6: Visualization of some convolution filters for the classification architecture

Model	Architecture
1	INPUT->[[CONV-> RELU]*2->POOL]*2 ->[FC-> RELU]*2 ->FC (CONV filter size 3)
2	INPUT->[[CONV->PRELU]*2->POOL]*2 ->[FC->PRELU]*2 ->FC (CONV filter size 3)
3	INPUT->[[CONV->PRELU]*2->POOL]*2 ->[FC->PRELU]*2 ->FC (CONV filter size 7)
4	INPUT->[[CONV->PRELU]*2->POOL->DROP]*3->[FC->PRELU]*2->DROP->FC (CONV filter size 3)
5	INPUT->[[CONV->PRELU]*2->POOL]*3 ->[FC->PRELU]*2 ->FC (CONV filter size 3)

Table 4: The different architectures used for the regression task

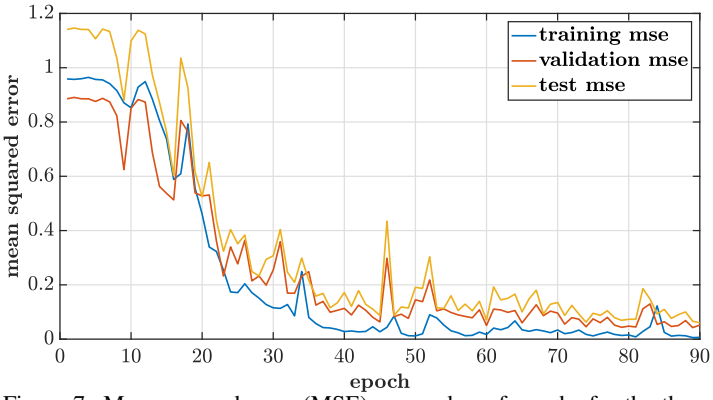

Figure 7: Mean squared error (MSE) vs number of epochs for the three parts of the dataset using the final regression model

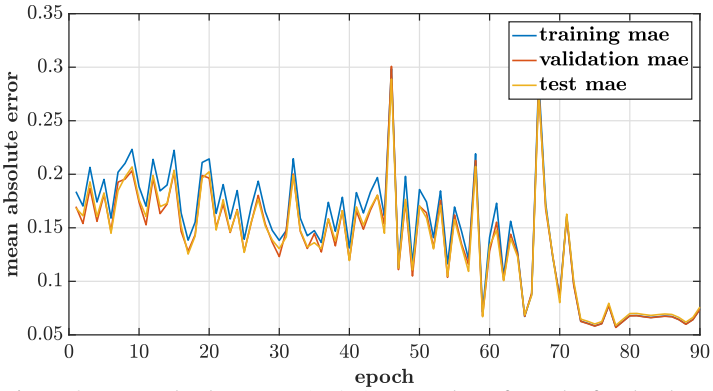

Figure 8: Mean absolute error (MAE) vs number of epochs for the three parts of the dataset using the final regression model

6 Smoothing using the Temporal Structure of the Data

From the input image, the CNN gives a noisy measurement of the steering, which could be harmful if implemented on a real system. We decided to exploit the temporal structure of the data to smooth the prediction from the CNN. The underlying structure of the system can be modeled as a Hidden Markov Model. The transition and observation distribution, however, are hard to come up with in this case because of the lack of a physical model. The CNN represents the sensor that outputs a measure of the actual steering command which is the hidden variable, hence this new prediction process assumed that the CNN has been pre-trained and does not involve learning the weights of the network anymore. In order to overcome the modeling issue we decided to simplify the structure as represented in figure 10.

Model	Training Loss	Validation Loss	Test Loss	Number of epochs
1	0.9572	0.8855	1.1410	6
2	0.0416	0.1242	0.1239	60
3	0.0304	0.1409	0.1542	60
4	0.1814	0.2887	0.6393	120
5	0.0066	0.0519	0.0601	90

Table 5: MSE performance for the different regression architectures

The main idea is that two consecutive steering command are unlikely to be very different, and that we should use this intuition in the prediction process. For this purpose we implemented two linear models: a first-order model using the previous command, and a second-order model using the two previous commands.

$$\text{First-order: } \hat{s}^t = \mathbf{w_1}\, s^t_{CNN} + \mathbf{w_2}\, s^{t-1}$$
$$\text{Second-order: } \hat{s}^t = \mathbf{w_1}\, s^t_{CNN} + \mathbf{w_2}\, s^{t-1} + \mathbf{w_3}\, s^{t-2}$$

Here, s^t_{CNN} is the CNN prediction at time t, s^{t-1} is the actual steering command that we experienced at time $t-1$, and $\hat{s}^t$ is the steering estimate. In order to find the weights $\mathbf{w_i}$ we can solve a simple linear regression problem using the actual steering command and their associated predictions. The objective function to minimize in the first-order case is:

$$\text{TrainLoss}(\mathbf{w}, s, s_{CNN}) = \frac{1}{m}\sum_{t=1}^{m}[\hat{s}^t - (\mathbf{w_1} s^t_{CNN} + \mathbf{w_2} s^{t-1})]$$

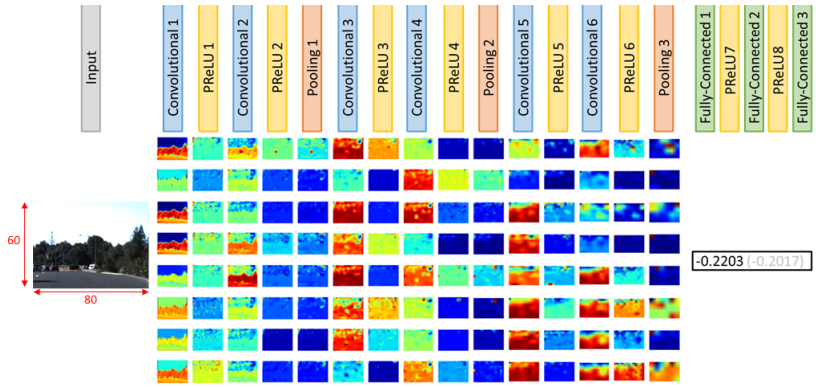

Figure 9: Visualization of some convolution filters for the final regression architecture

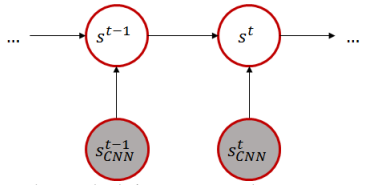

Figure 10: The underlying temporal structure of the data

Model	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$
First-order	0.0420	0.9535	N/A
Second-order	0.0165	1.829	-0.8482

Table 6: Weights of the linear temporal models

RMS of CNN predictions	0.203156
RMS of first-order smoothing	0.488451
RMS of second-order smoothing	0.284186

Table 7: RMS of the different models

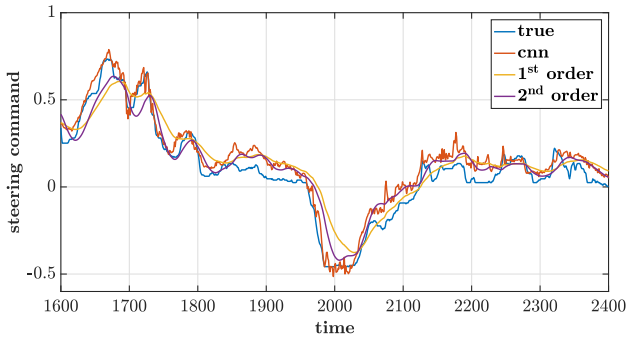

Figure 11: Comparison of the CNN predictions to the smoothed ones

Exploiting the temporal structure smoothed our predictions. The new predictions are dominated by the preceding time steps and therefore lag behind the actual steering command, while the pure CNN predictions are noisier but follow changes in the true steering commands more promptly. The root-mean-square deviations to the true steering commands as presented in table 7 confirm this observation

7 Conclusion and Future Work

The results show the capability of Convolutional Neural Networks to learn complicated tasks from images. We were able to achieve reasonable performance in predicting the steering (less than 0.1 MSE on the test set) for the best architecture. Moreover, the first part was only a supervised learning task and we did not use any temporal information. The last section of this report showed how we can smooth the predictions using this information. This smoothing step could be very interesting if we were to implement the algorithm as a controller on a real car. However, many challenges still need to be addressed to come to this point. Indeed, our network was trained on a short sequence of images (roughly 10 minutes of driving) with the same traffic and weather conditions. To make sure that the network is able to generalize well even in unseen environment we could use data augmentation techniques as they used in [3]. Another interesting addition to the project could be to add the temporal structure during the training. Using Recurrent Neural Networks is a possible follow-up, or finding a way to combine the Hidden Markov Model described in section 6 with the CNN algorithm.

References

[1] Daniel V McGehee, Mark Brewer, Chris Schwarz, and Bryant Walker Smith. Review of automated vehicle technology: Policy and implementation implications. 2016.

[2] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[3] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[4] Udacity. Open sourcing 223GB of driving data collected in Mountain View, CA by our Lincoln MKZ. `https://medium.com/udacity/open-sourcing-223gb-of-mountain-view-driving-data-f6b5593fbfa5#.a9fdmlkk5`, 2016.

[5] Fei-Fei Li, Andrej Karpathy, and J Johnson. CS231n: Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/convolutional-networks/`, 2015.

[6] F.Chollet. Train a simple deep cnn on the cifar10 small images. `https://github.com/fchollet/keras/blob/master/examples/cifar10_cnn.py`, 2015.