TSFS12 HAND-IN EXERCISE 5
Learning for Autonomous Vehicles

October 1, 2020

1 OBJECTIVE

The objective of this exercise is to investigate different aspects of learning for autonomous vehicles. More specifically, regression of driver path data will be performed using Gaussian processes and reinforcement learning will be studied in a scenario with a grid world with a moving autonomous vehicle, modeled as a particle. In the extra assignment for higher grades in Appendix A, the task is to investigate neural networks for making predictions of lane changes on highways. Compared to the previous four hand-in exercises, most of the implementation code required is provided in this exercise. The intention of the exercises is to provide an understanding and hands-on experience with the methods discussed during the lectures on learning for autonomous vehicles.

2 PREPARATIONS BEFORE THE EXERCISE

Before performing this hand-in exercise, please make sure that you have read and understood the material covered in:

- Lectures 10–11.
- Sections 2.1–2.3 in Rasmussen, C. E., & C. K. I. Williams: *Gaussian Processes for Machine Learning*. MIT Press, 2006.
- Sections 4.1–4.4 and 6.1–6.5 in Sutton, R. S., & A. G. Barto: *Reinforcement learning: An introduction*. MIT Press, 2018.

To get code skeletons to be used for the different exercises, download the latest files from `https://gitlab.liu.se/vehsys/tsfs12` and familiarize yourself with the provided code. The data needed for the exercise on Gaussian process regression and the extra assignment on neural networks are available in Lisam under
`Course documents/i80_data`
Please download the whole directory and save it on your computer, and then unzip it in the same directory as the remaining files. In the student labs, the data can be found at `/courses/tsfs12/i80_data` and you do not need to download the data (about 380 MB in total).

You are free to choose in which language to implement these exercises. Code skeletons are available in Matlab and Python (though only Python for the extra assignment for higher grades), but if you prefer to make your own implementations that is also possible. This document is written with reference to the Matlab files, but pointers towards corresponding Python equivalents are provided in the text. In the student labs, all packages needed are pre-installed in the virtual environment activated by

```
1 source /courses/tsfs12/env/bin/activate
```

3 REQUIREMENTS AND IMPLEMENTATION

The objective of the exercise is to get a practical understanding of different methods for learning for autonomous vehicles and evaluate their properties in some example scenarios. Therefore, to pass this exercise you should solve the following tasks:

- Perform path regression for driver data from a set of trajectories from a highway section, and visualize the results in terms of mean value and variance. In addition, investigate a simple method to investigate class association for different input paths.
- Apply value iteration to solve a Markov decision process in a grid-world scenario under known environment dynamics.
- Apply Q-learning to the same problem as in the previous task, but with *a priori* unknown state-transition probabilities.

The exercise is examined by presenting your solution to one of the teachers in the course over Zoom. Time slots for presenting your solutions will be announced in Lisam. During the presentation you should be prepared to answer the questions stated in connection with the different tasks in this document. In addition, the following should be submitted on the Lisam course page by the deadline:

1. Runnable code. It is *not* required to submit code that automatically runs all the cases of parameter variations for the different methods. If your implementation consists of several files, submit a zip-archive.

See Appendix A for the extra assignment needed for higher grades. The extra assignment is performed individually and is submitted in Lisam by a document answering the questions stated, including suitable figures and plots. There is only pass/fail on the extra assignment and the document can not be revised or extended after the first submission.

> Since these exercises are part of the examination of this course, we would like to ask you not to distribute or make your solutions publicly available, e.g., by putting them on github. (A private repository on gitlab@liu is of course fine).
>
> Thanks for your assistance!

4 DISCUSSION TOPICS & EXERCISES

This hand-in exercise is divided into two different parts, with sub-tasks to be performed for each. The required implementation code for solving the different tasks is to a large extent provided for the exercises in the skeleton files.

4.1 Regression of Path Data Using Gaussian Processes

In this first part of the hand-in exercise, regression based of path data from drivers in a busy highway section will be investigated. The method that will be used is based on Gaussian processes for the modeling. The method used in the exercise is a simplified version of the method presented in the following paper:

- Tiger, M., & F. Heintz: "*Online sparse Gaussian process regression for trajectory modeling*". International Conference on Information Fusion (FUSION), pp. 782–791, 2015. (https://www.ida.liu.se/divisions/aiics/publications/FUSION-2015-Online-Sparse-Gaussian.pdf)

The data used in this exercise are recorded trajectories of vehicles on a U.S. highway, I-80 in Emeryville, California[1].

The traffic site is shown in Figure 1a, and it is a six-lane highway with an on-ramp, with the high-speed lane on the left. In the data set there are 4 383 recorded trajectories over 3 times 15 minutes. In Figure 1b, 250 trajectories are plotted over the 500 m long highway section.

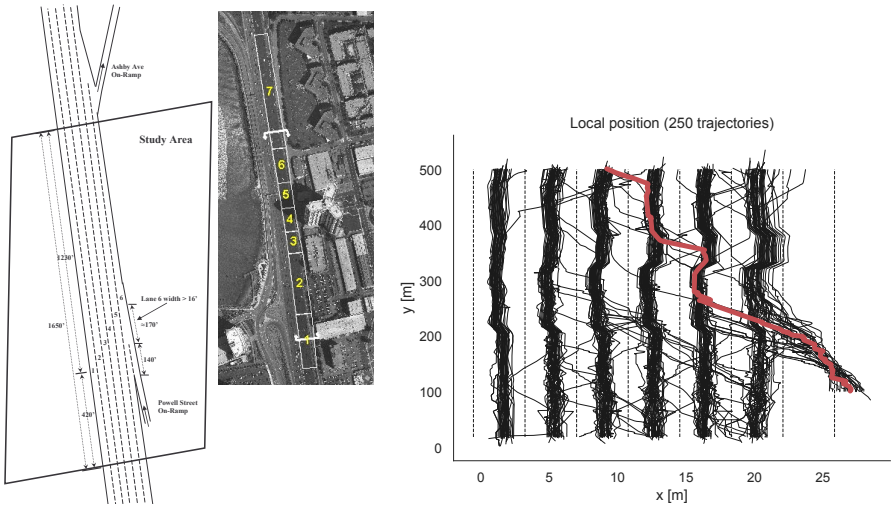

Figure 1: I-80 at Emeryville, California, extracted from the data set from the U.S. Department of Transportation data hub described in the text.

In the provided code skeleton, it is assumed that the zip file with data mentioned above is extracted in the same directory as where the files for the hand-in exercise are located. If you are in the student labs at campus, you can just uncomment one line in the code skeleton to get the correct directory reference. The data set contains several different driving paths, comprising different lane changes as well as merging from the on-ramp. In Matlab, the lanes are numbered from 1–6, left to right, and the on-ramp is denoted 7. In Python, the lanes are numbered from 0–5, left to right, and the on-ramp is denoted 6.

Exercise 4.1. The file `driver_path_regression_I80.m` contains code for visualizing the driver paths in the highway section for one of the three data sets (that between 05:00 and 05:15). In Python, there is a corresponding notebook called `driver_path_regression_I80.ipynb` with the equivalent implementation. Investigate the data set to get an understanding of the scenario and the information contained in the data set.

Now we will use the driver data to make Gaussian process (GP) models for some of the scenarios observed in the data set. We will consider different subsets of the data, corresponding to paths with a specified initial lane and final lane. These are defined in the variables `init_lane` and `final_lane`. Here, we will focus on the paths starting at the on-ramp (i.e., `init_lane=7` in Matlab and `init_lane=6` in Python) and ending up in another lane 1–6 (Matlab) or 0–5 (Python).

Since the different driver paths are of different length, the independent variable for all paths are computed as the normalized length (such that $s \in [0, 1]$ for all driver

1 I-80 data set citation: U.S. Department of Transportation Federal Highway Administration. (2016). Next Generation Simulation (NGSIM) Vehicle Trajectories and Supporting Data. [Dataset]. Provided by ITS DataHub through Data.transportation.gov. Accessed 2020-09-29 from http://doi.org/10.21949/1504477. The data are open source and on this link you can also find additional information about the data set.

paths). This normalized variable is called s, since it can be interpreted as a path coordinate. The code for performing this procedure is available in the skeleton file. Linear interpolation between the available path points is used to get values for the x and y coordinates for arbitrary values of the path coordinate s.

To perform the Gaussian process regression, we will use a toolbox called GPML in Matlab and a module in `scikit-learn` in Python. The toolbox for Matlab can be downloaded from

- http://www.gaussianprocess.org/gpml/code/matlab/doc/

If you would like to investigate Gaussian processes more in detail, there are also several examples with associated code on this page that you can study before continuing with this exercise. The Python package `scikit-learn` can be installed through the major package managers for Python installations. This package also has a homepage with several examples of Gaussian process regression:

- https://scikit-learn.org/stable/modules/gaussian_process.html

Now the task is to determine two GP models, one for the x-coordinates and one for the y-coordinates, both with s as the independent variable. The training data consisting of `N_samples_gp = 8` randomly chosen points along each driver path are used to optimize the hyperparameters in a squared exponential covariance function. The optimization of the hyperparameters is done using the function `minimize` in the Matlab toolbox. In the provided code, there is a wrapper function called `gp_reg_hp_fit` that is used for the hyperparameter computations. In Python, equivalent functions from the `scikit-learn` module are used.

The actual prediction of output values for input values of s that are different from those used for the training is then performed using the function `gp_reg_pred`. In Python, equivalent functions from the `scikit-learn` module are again used.

The code for performing the optimization of the hyperparameters, i.e., the parameters σ_f, l of the chosen covariance function and the noise level σ_ε of the output data, is provided in the skeleton file. In addition, the code for performing prediction based on the GP models for new inputs are provided.

Exercise 4.2. Read through the provided code for the hyperparameter optimization and GP model prediction and make sure that you understand the different steps and the respective output variables. Execute the code for performing the fit of the hyperparameters of the GP models to the training data. Study the numerical values of the hyperparameters. What do the different hyperparameters correspond to?

Exercise 4.3. Plot the training data and the predicted mean for the respective model in two different plots (one for x and one for y). Also plot a confidence interval for the prediction, by using the computed variances (e.g., by using two standard deviations, where the standard deviation is the square root of the variance).

Exercise 4.4. Visualize the mean of the predicted x and y values by plotting them (y as function of x) in the same figure as the highway lanes and the driver paths used for the training. In addition, visualize the uncertainty by plotting ellipses at selected path coordinates along the predicted mean values. To plot an ellipse with the half axis determined as two times the standard deviations along the respective direction, the following code can be used (in Matlab syntax, for path coordinate index `i`):

```
phi = linspace(0,2*pi,50);
plot(2*sqrt(path_x_var(i))*cos(phi),2*sqrt(path_y_var(i))*sin(phi))
```

The ellipse can be translated in the xy plane to the correct position by adding the mean values to the respective plot argument.

Are the predicted means from the GP models in good agreement with the training data? Are the estimated variances expected, given the variation of the training path data for the respective GP model?

Exercise 4.5. Keep the initial lane the same, i.e., starting at the on-ramp. Change the final lane (using the variable `final_lane`) and compute and visualize GP models also for these scenarios. Are the estimated GP models in agreement with the data? How do the different number of paths available for different combinations of initial and final lanes affect the resulting GP models?

Finally, we will use the estimated GP models to investigate if the models can be used to compute a significant indicator of which class different paths belong to (i.e., which initial and final lanes a certain path starts and ends in). To this purpose, we will use a score function, measuring the fit of a particular output sequence from the GP models to the actual outputs from the data set. By computing the score function values for paths from different final lanes (but all starting at the on-ramp, number 7 (Matlab) or 6 (Python)) with the GP model for a specific scenario, we can compare the score functions to evaluate which class the paths is most likely to belong to. This is performed in the function `predict_scenario_score`. In the skeleton file, there is code available for computing this and visualizing the results. To get quantitative values for comparisons that we can interpret as probabilities, we apply the softmax transform (see function `softmax`) to the predicted scores.

Exercise 4.6. Read through the code provided for the prediction of scores and probabilities for class association. Test different scenarios (i.e., different final lanes) and investigate if you get significant results for the probabilities. Are those paths that give higher similarity of scores, also similar in some sense when it comes to the actual paths?

4.2 Reinforcement Learning Using Value Iteration and Q-Learning

In this part of the hand-in exercise, we will investigate two different algorithms for solving a Markov decision process using reinforcement learning. More specifically, we will investigate value iteration under known environment dynamics as well as Q-learning in the more complex case of unknown state-transition probabilities. The assignment is based on

- Example 6.6 in Sutton, R. S., & A. G. Barto: *Reinforcement learning: An introduction*. MIT Press, 2018.

The scenario is an autonomous vehicle, here modeled as a particle, moving in a grid world from state to state (see Figure 2). In each state in the grid world, the vehicle has four different actions to choose from: left, right, up, and down. These are counted as number 1–4 (in Python 0–3) in the implementation, in the respective order. Because of disturbances acting in the modeled world, applying a certain action does not necessarily mean that the vehicle ends up with moving in the intended direction; instead there is a certain probability that the vehicle will move sideways compared to the intended motion. If the vehicle applies an action that would bring it outside of the world, it remains in the same state and once again receives the reward associated with that state. The overall task is to maximize the expected accumulated reward for the vehicle from start to a terminal state.

The motion starts in the lower left corner (S) and the goal is to move to the lower right corner (G). There are certain states in the world that are occupied by obstacles, and the states between the start and goal state in the lower row comprise a cliff where the vehicle gets stuck. The cliff states and the goal states are here considered as terminal states.

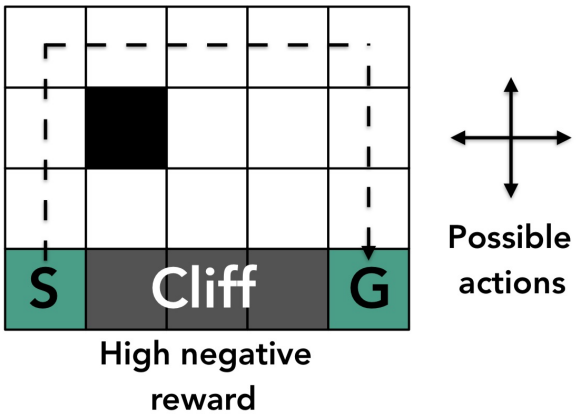

Figure 2: The grid world where the autonomous vehicle is moving from start (S) to goal (G).

The reward associated with the cliff states is selected as a large negative number. The reward for the goal state is chosen to be zero and the remaining free states (i.e., states not occupied by obstacles) have a small negative reward associated with them. The latter choice is to make it cost to take (unnecessary) steps in the world before reaching the desired goal state, which can be interpreted as trying to keep the path length as short as possible, though under consideration of the risk of getting stuck in the cliff states.

The code required to perform the exercises in the following is available in the files named `rl_val_iter.m` and `rl_q_learning.m` (a smaller update is required in the file called `rl_q_learning.m` in one of the exercises). The parameters defining the scenario are stored in the struct `params`, and the properties of the parameters are described in the commented code. In Python there are notebooks with corresponding content called `rl_val_iter.ipynb` and `rl_q_learning.ipynb`. In the Python version, a subset of the auxiliary functions for this task are available in the file `rl_auxiliary.py`.

Exercise 4.7. Read through the code provided in the file `rl_val_iter.m`, and study in particular the variables available in the struct `params`. Then execute the script for the default values of the parameters. For each iteration, the current value function and the associated actions are shown in two different plots. The script is paused between each iteration, and the execution is continued by pressing enter on the keyboard. Run the script until convergence. Which is the optimal path from start to goal? Note that the solution does not only provide this path, but an optimal policy for all states.

Exercise 4.8. Now we will investigate how changes in the state-transition probabilities affect the resulting optimal policy. Vary the probability of the car to be subject to disturbances (parameter `P_move_action`, in the interval (0,1]). What happens if you set this parameter to 1 and what does it mean in practice?

Exercise 4.9. Vary the discount factor (parameter `gamma`) and study the results on the optimal policy. Do you see any noticeable effects, and are the results in agreement with intuition?

Exercise 4.10. Finally, vary the negative reward received for ending up in the cliff states (parameter `R_sink`). How does it influence the optimal path from the start state to the goal state, if it is made significantly less or more negative?

Now we will investigate the case when the state-transition probabilities are not known *a priori*. The same scenario and Markov decision process as in the previous exercise will be solved, but now using Q-learning instead. The learning rate is determined by

the parameter `alpha` and a total number of `nbr_iters` iterations are performed. The trade-off between exploration and exploitation during the iterations is decided using the epsilon-greedy strategy, where `eps` is the parameter governing the trade-off. This strategy is implemented in the function `select_eps_greedy`.

Exercise 4.11. Read through the code provided in the file `rl_q_learning.m`, and study in particular the variables available in the struct `params`. Then execute the script for the default values of the parameters (we here start with the case where there are no disturbances and the vehicle thus always moves according to the specified action, i.e., `P_move_action = 1.0`). In contrast to the value iteration, only the final resulting value function and corresponding actions are shown in the plots since there are typically many more iterations required for Q-learning with unknown state-transition probabilities. In addition, a plot is shown where the averaged accumulated rewards for the episodes are visualized over the iterations. Does the Q-learning converge to the same optimal policy as the value iteration for the same set of parameters?

Exercise 4.12. Vary the probability of the car to be subject to disturbances (parameter `P_move_action`, to something slightly lower than 1). How does it affect the Q-learning? Does the Q-learning find the same optimal policy from start to goal as the value iteration for the same set of parameters? How about the policy at the states other than the start state?

Exercise 4.13. An important parameter in the Q-learning is the parameter `eps` governing the exploration. Typically, it is desirable with more exploration in the beginning of the iterations, and then in the end you rely almost completely on the information you have already acquired. Update the implementation such that the parameter `eps` is made dependent on the iteration number, and successively decreases towards zero, starting from one, when the iteration number increases. The iteration number `k` is available as an input to the function `select_eps_greedy`. Do you see any improvements in the performance of the Q-learning and its convergence to the optimal policy?

A EXTRA ASSIGNMENT

Short-Term Prediction of Vehicle Behavior Using a Neural Network

Knowing where you are and in which environment, i.e., localization and mapping, is essential for an autonomous vehicle. A next step is to know what your environment will look like in the future, i.e., predicting the future motion of your surrounding vehicles is a critical step that is necessary for safe and reliable motion planning [2]. Modeling human behavior, in this case driving behavior, is difficult and one interesting option is to utilize recorded data to build models of behavior. In this exercise you will design and experiment with a model of short-term vehicle behavior using a neural network.

The data set that will be used as a basis for the models are the recordings from the I-80 highway section that were used in the Gaussian process regression exercises earlier. In that exercise, only parts of the data were used, whereas here we will use more data from this set to make lane-change predictions.

How to predict vehicle motion? On a very short time-scale, vehicle velocity and inertia is sufficient to estimate where the vehicle will be in the immediate future. However, on a time-scale beyond roughly one second, drivers start to act and interact with its surroundings, e.g., change lanes, and inertial prediction becomes less accurate. Learning approaches have been shown to be effective for prediction on a three-second horizon [1]. The objective of the exercise can therefore be summarized as

> *Make a model that predicts if a vehicle will shift to the lane on the left, on the right, or continue in the same lane for the next 3 seconds.*

There are many things that can influence how a driver chooses lane and indicators of impending lane change. Naturally, positions, velocities, and accelerations of surrounding vehicles, the same lane and the neighbouring lanes in relation to the ego vehicles position, velocity, and acceleration are main factors. But these are not the only ones, for example vehicles more often tend to aim for the fast-moving lanes, but this also depends on the density of vehicles in each lane and of course the (unknown) destination. To make a rule-based model of lane-change prediction, weighing all of these factors is non-trivial and one possible alternative is to use black-box models and data to model driver behavior. In this exercise, neural networks will be investigated as a tool for accurate lane-change prediction. Recommended reading are the lecture notes and the associated reading material.

Hint: If you are new to neural networks, the web-based Tensorflow Playground platform available at `https://playground.tensorflow.org` is a good place to start experimenting and familiarize yourself with the concepts before solving this exercise. You are also encouraged to look around in the introduction `https://www.tensorflow.org/learn`.

For implementation of (deep-)neural networks it is recommended to use any of the well established tools like Tensorflow or PyTorch. We do not recommend to use Matlab for this, so this exercise is only available in Python. However, there will be very little code to write, a fully running skeleton is provided, so it is still possible to do this exercise without any prior python knowledge. The skeleton is written for Tensorflow using the keras high-level API to Tensorflow. You are, however, welcome to use any framework you like. The models you will develop here will be small enough to run on almost any computer, so access to, e.g., GPU resources is not necessary. However, the provided code runs unchanged on a GPU so try that if you have access to such.

To run the Python code provided, you need some additional Python packages: pandas, scikit-learn, and tensorflow. All can be pip-installed on your own computer and

they are pre-installed at the university Linux-labs in the virtual environment that is activated by

```
% source /courses/tsfs12/env/bin/activate
```

In this exercise you will experiment with a very basic approach, if you are interested in this topic you can find more in recent research, e.g., works from Uber Research in [1, 2].

a.1 Data and Model Description

Pre-processed data are prepared for this exercise and can be downloaded from the directory `Course documents` in Lisam. This includes pre-processed features and the whole set of vehicle trajectories, ready to be used for learning your predictive model. If you have not downloaded the complete data set already, download the whole `i80_data` directory, unzip it, and place the directory in your code directory. In the student labs, the data can be found at `/courses/tsfs12/i80_data` and you do not need to download the data (about 380 MB in total).

For each recorded trajectory, a feature vector x and a label y is recorded every 3 seconds. Thus, for a 24 seconds long trajectory, we have 8 data points in total for that trajectory. The label $y \in \{0, 1, 2\}$ and is interpreted as

$$y = \begin{cases} 0 & \text{if the vehicle will change to the left within three seconds} \\ 1 & \text{if the vehicle will stay in lane for the next three seconds} \\ 2 & \text{if the vehicle will change to the right within three seconds} \end{cases}$$

This means that the variable y is what we want to predict. The feature vector x consists of 41 values. There are 6 surrounding vehicles tracked, in front and behind in the current lane and the two adjacent lanes. For each of these 6 vehicles, velocity and acceleration and the time and space distances are used as features. This gives $6 \times 4 = 24$ features. Then for each of the 6 lanes, mean velocity and density are used and that gives another $6 \times 2 = 12$ features. Finally, for the ego vehicle, the lane number, previous lane number, velocity, acceleration, and relative position within the lane is recorded, which gives another 5 features. This in total gives the 41 features. The model is then a function f such that

$$\hat{y} = f(x) \tag{1}$$

where $\hat{y}$ is the predicted action by a vehicle at some time-point and we want to find the predictor in (1) such that $\hat{y}$ predicts the true labels y as accurately as possible. The feature vector x is summarized in the file `features.md`.

The complete data set consists of 4 383 trajectories resulting in 95 591 data points, where 963 are left-lane changes, 304 right-lane changes, and 94 324 straight ahead. This strong imbalance in data, where the vast majority of data points correspond to straight-ahead motion, is important to handle, otherwise a learned predictor risks to be severely biased. A basic way to do this is to weight underrepresented classes corresponding to the imbalance. Here, balancing of data is done by creating a training data set with M (a parameter to choose) samples from each class. Here, we want M to be significantly larger than the size of the smallest class (304 right-lane changes) so the sampling is done *with replacement*. Choosing the value of M is something you will experiment with during the exercise. There is also a step where data are separated into training and validation data sets. In the notebook, you will get data arrays in the variables

- `x_train` and `y_train` – training data points and class labels
- `x_val` and `y_val` – validation data points and class labels

that are ready to be used by Tensorflow to estimate a model. There is also a normalization step of `x_train` and `x_val`, such that each feature has mean value 0 and standard deviation 1 in `x_train`. Do not forget to apply this normalization also when predicting!

a.2 Exercises

There is a skeleton notebook `extra.ipynb` that you can use as a starting point and the objective is for you to experiment with all hyperparameters to find a model and evaluate the performance. You do not need to elaborate with advanced neural networks, basic structures are sufficient. The problem to predict vehicle behavior is not easy, so you can not expect 99 % accuracy in your model, but performance around 80 % is possible to achieve.

Exercise A.1. Explain why the imbalanced data set is a problem and what would happen if no measures were taken, e.g., the resampling step implemented in the code skeleton?

Exercise A.2. The command `model.summary()` summarizes the model and also prints out the number of trainable parameters in the model. For a chosen model, preferably your final choice of model, show the output of `model.summary()` and explain the number of parameters in the model.

Exercise A.3. Experiment with all hyperparameters in the model, e.g., number of resampling points in the balancing step, number of data points chosen for the validation data, and architecture of the model (number of layers, number of nodes in each layer, regularization parameters, ...).

Plot the evolution of loss function and accuracy, for both the training set and the validation set, during training. You can expect results similar to what is shown in Figure 3.

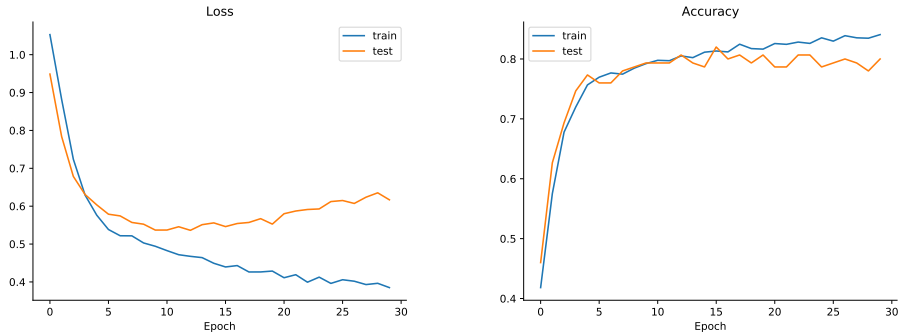

Figure 3: Loss and accuracy of training and validation data during training.

Report on your findings and comment on what the plotted curves mean.

Exercise A.4. Compute a confusion matrix for the validation and training data using the function `sklearn.metrics.confusion_matrix` (already imported in the skeleton file). Explain the output.

Exercise A.5. Code to visualize predictions on a trajectory is included, resulting in plots like Figure 4. Predictions in the plot are made at the red dots and the

corresponding colors represent predicted action in the next three seconds. Plot similar figures for trajectories in the validation set and explain what you see. Discuss strengths and weaknesses of your model.

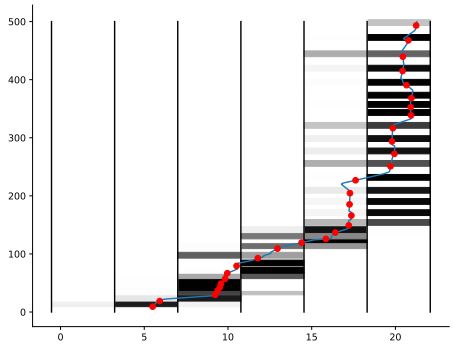

Figure 4: Lane prediction for a specific vehicle. Note that surrounding vehicles are not included in the figure. A darker color indicates probable lane during the next three seconds.

Exercise A.6. If you have experience with other machine-learning tools or methods, you are encouraged to compare the performance of your neural network with other classifiers. The package scikit-learn (`https://scikit-learn.org/`) has a number of classifiers out-of-the-box applicable to the data, for example support vector machines and random forest classifiers.

REFERENCES

[1] Nemanja Djuric, Vladan Radosavljevic, Henggang Cui, Thi Nguyen, Fang-Chieh Chou, Tsung-Han Lin, Nitin Singh, and Jeff Schneider. Uncertainty-aware short-term motion prediction of traffic actors for autonomous driving. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, March 2020. https://openaccess.thecvf.com/content_WACV_2020/papers/Djuric_Uncertainty-aware_Short-term_Motion_Prediction_of_Traffic_Actors_for_Autonomous_Driving_WACV_2020_paper.pdf.

[2] Sai Yalamanchi, Tzu-Kuo Huang, Galen Clark Haynes, and Nemanja Djuric. Long-term prediction of vehicle behavior using short-term uncertainty-aware trajectories and high-definition maps, 2020. https://arxiv.org/abs/2003.06143.