Report first assignment Deep Reinforcement Learning
by Nina Braunmiller k11923286

Behavioral Cloning

- As model that should learn via its parameters was implemented a neural network which was used as the basis of my policy. Because we work with image data taking convolutional layers, three in total, made sense to extract important features of the images. Then a flattening layer was implemented with the aim to not lose information but transforming the convolutional output to a suited input for fully connected linear layers as classifier part of the network. Two hidden linear layers further used parameters to learn. The output layer returned the number of neurons same as the number of possible actions. Softmax() was **not** used after oberserving, that it results in bad performance of the model.
- Initialize model parameters: This was done in the **function initialize_weights_net().** Here I was inspired by LeCun's initialization. The approach initializes weights with a normal distribution with mean of 0 and variance of 1/J where J is the number of incoming connections for a single neuron. I assumed that a single neuron in convolutional layer is the resulting feature map as it also has only one incoming bias unit. Also the bias units were considered as part of J. This was done manually for convolutional and linear layers as you can see in the relating function.
- The hyperparameter learning rate: My idea was it to use an optimizer which adapts the learning rate to the underlying data using PyTorch‘s torch.optim.Adadelta(). Adadelta some kind of normalizes the added gradient when updating parameters but also considers amount differences between the past updates such that high differences lead to a higher current learning rate.
- The Behavioral Cloning process was implemented by feeding batched states into the network in a supervised setting. I used the Cross Entropy Loss as loss function to learn via gradient descent by using the backward() function of PyTorch. I have chosen this loss function because we have a discrete action space.
- Training the net on ten epochs lead to a submission score of 646.576. By simply extending the number of epochs up to twenty the final result was 678.861.

*DAgger algorithm (**wasn‘t used for the submission!**)*

- Here I have to say that I didn‘t use its results for submission but implemented it. I could not observe a benefit in using it. Maybe it‘s a question of hyperparameters. It is probable that I used to less outer loops such that there were simply too few new states on which the already trained classifier had to learn on. Besides that, I actually have a question for you. When creating a trajectory do I always start at the fixed starting point or do I change the starting point for each trajectory? I think, the second one makes more sense.
- I started with an empty dataset in the outer loop which was with each loop extended by the return of the inner loop, the dagger() function. On this dataset I trained the already trained model even further via behavioral cloning again.
- The created trajectories of the inner loop had the length of one epoch each. The hyperparameter beta were tried out with values of 0.5 and 0.7 which determined whether the given expert policy or the already trained behavioral cloning policy were used to build the trace. To each state a label given by the expert policy were assigned. For implmentation have a look at the function dagger().