Project Milestone#2: TRAIN

Junkawitsch
s8hejunk@stud.uni-saarland.de

2. Exercise Assignment

(I.1)

I also present the README file here for reference and basic understanding of my framework.

This is the Karel Task neural network training framework.

The **'/models'** folder contains 2 neural network model:

- mlp.py contains a simple Multilayer Perceptron which I used for debugging and and early development.
- cnn.py contains the final neural network model. It uses convolutional layers in the shared part of the network and 2 mlps for the value network and policy network.

The **actions.py, direction.py, and environment.py** modules define the environment of the Karel task:

- action.py defines the 6 actions the agent can take and some basic functions.
- direction.py is a simple module that takes care of the agent's direction on the grid.
- environment.py is the main environment module that define the state of our reinforcement learning task.

The **dataset.py, rewards.py, sequences.py, and train.py** are the core modules of the training framework:

- dataset.py defines a custom dataset that i can use with a dataloader for efficiently loading the training tasks and validation tasks in batches.
- rewards.py is a simple module that defines the reward function.
- sequences.py defines the Sequence class for our optimal training sequences.
- train.py is the core training loop and training setup containing the backward pass and episode enrolling functionality. (Actor-Critic)

The **utils.py** module contains some basic utility functions.

The **logger.py, test.py, validate.py, and visualize.py** are used for evalating and documenting training runs:

- logger.py is used during the training to log different learning values (loss, correctness, time, etc.)
- test.py evaluates a model using a single task from a data set.
- validate.py evaluates a model on a whole data set (using batches).
- visualize.py can be used to plot graphs of the logs created with logger.py.

Preprint. Under review.

(I.2)

a) Feature Representation

The state of the reinforcement learning task is defined in the environment.py module. The state class has a function that creates the feature representation for a specific state (current grid configuration and goal grid configuration). (def get_feature_representation(self)).

The feature representation almost matches with the proposed feature representation of the first milestone except for the wall matrix. Since the walls are *always* exactly identical in the pre- and post-grid we obviously only need one matrix for the wall configuration.

Therefore I have an input tensor of $b \times 5 \times 4 \times 4$ where b is the batch size.

This feature representation makes it easy to use convolutional neural networks since I directly provide the 2-dimensional data to my network model. If I instead want to use a classical Multilayer Perceptron I can include a Flatten layer in the network before I use the simple fully-connected layers.

However a convolutional neural network has proven to provide the best results.

b) RL algorithm

For the Reinforcement Learning algorithm I obviously chose a gradient policy algorithm. I directly went for the Actor-Critic algorithm since the e-mail discussion stated that we should use the more advanced AC algorithm to train the Agent on the big data set.

The first thing was to get a deep understanding of the One-Step Actor-Critic on page 332 in the Sutton and Barto Reinforcement Learning book. The algorithmic box shows an online learning approach meaning that we do not enroll a complete episode before updating the state value parameters and policy parameter. Instead we choose one action after another and update the parameters after every step according to the update rules. I implemented this online approach using batches but did not get any good results. The neural network was not able to solve any tasks in the easy data set.

Consequently I googled and did some research on the internet and came across a github repository that was also mentioned in the e-mail discussion. (https://github.com/pytorch/examples/blob/master/reinforcement_learning/actor_critic.py) Instead of using the online approach their approach first enrolled *one* complete episode and afterwards updated the parameters of the neural network.

I implemented a similar approach but including batches for faster learning and better GPU utilization. This required some rather complex tensor computations.

My final solution loads a batch of tasks and optimal sequences into memory and simultaneously enrolls every task till it either crashes or finishes. It saves the rewards, log probabilities, state values, and actions. After every episode is completely enrolled I perform one big backward pass through the network, updating the parameters based on the rewards, log probs, and state values.

The loss function was rather complicated to define and required a bit of research.
(https://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_5_actor_critic_pdf)
The website https://medium.com/geekculture/actor-critic-implementing-actor-critic-methods-82efb998c273 stated that the update rule of the state value parameters basically represents the MSE loss, although in my testing the MSE loss performed a lot worse than the smooth_l1_loss used in the previously mentioned git repository. Therefore I decided to stick to the smooth l1 loss that basically combines the l1 and l2 loss.

c) Neural Network architecture

There are two ways of using neural networks with the Actor-Critic algorithm. We could use two completely seperate neural networks with no shared weights for the state values and the action probabilities or we could combine the state value network and action policy network to make use of shared weights.

I went for an intermediate approach. I designed one neural network that first uses three 2-dimensional convolutional layers with a 3×3 kernel a padding of 1 and a stride of 1. The first convolutional layer increases the number of channels from 5 to 7 to increase the amount of feature maps we learn in the shared part of the network. The goal of the shared part is to learn a reasonable and useful representation of the state feature representation.

The 7 learned 4×4 feature maps are then used in two seperate MLPs to predict the state value and the six action probabilities. The two MLPs are completely identical and contain 4 fully-connected layers with ReLU activation functions after every layer. The amount of neurons in each layer is decreasing: $7 \cdot 4 \cdot 4 \to 80 \to 40 \to 20 \to 6 \wedge 1$.

I finally used the softmax activation function to get correct probabilities for the six actions.

I also provide the exact network architecture in the last section.

d) Agent's hyperparameters

Important hyperparameters and information:

- batch_size $= 256$
- $\gamma = 0.97$
- max_steps $= 5/(1-\gamma)$
- learning_rate $= 0.0001$
  The learning rate is not that important since I used the Adam optimizer that adaptively changes the learning rates based on estimates of the first and second moments of the gradients.
- optimizer $=$ Adam optimizer
  I also tried the simple SGD optimization algorithm, however this optimizer worked a lot worse than the more advanced optimizer Adam. This is most likely due to the fact that Adam adaptively changes the learning rate for each parameter. This is especially useful if we notice that we combined the state value network and policy network into *one* network meaning that the two network parts may need different learning rates.
- Every neural network is trained on my Nvidia RTX 2070 Super to get the best training times and save some power since my GPU is only at around 9% usage.

Of course there are a lot more minor hyperparameters and design choices that I already mention in previous sections.

(I.3)

Curriculum Design

In my final network training runs I did not really need curriculum learning since I used other techniques like imitation learning and a simple reward design with large training batches and the complete rollout of the episodes. This already resulted in a good learning speed and accuracy of the trained model.

However my framework supports training one neural network model with different data sets. I also used this during development and debugging. Saving the current model, optimizer, and epochs allows me to first train the model on the easy data set, save it, load it again, and continue training with a more complex data set.

Reward Design

I already proposed a reward design in my first milestone. However I changed it a little bit to improve training. The immediate reward of a crash is 0 in my final training solution. If we penalize crashing the agent just tries to "survive". This may result in the agent never moving since moving has a high risk of crashing (moving into a wall or out of the grid). The final reward for a correct finish is 10. This is a rather arbitrary number and other values work aswell. 10 leads to nice loss values during training.

Finally, my reward design is quite simple but effective. I did not implement something like landmarks or other advanced reward function because they might result in the agent not learning to actually *finish* the task. So only giving a reward when correctly finishing ensures that the agent's goal is actually finishing the task. The discount factor ensure that the agent tries to finish as fast as possible.

Imitation Learning

Imitation Learning is *the* most important training technique. Without imitation learning the learning on the big or medium training data set is *not* possible. Instead of using the predicted action I use the optimal action to enroll the episode. The log_probability is based on the optimal action and the predicted action probability distribution. My framework allows easily switching imitation learning off and on. A few more lines of code would also allow to switch imitation learning off and on during the training. This would be handy if learning without imitation learning is stuck and needs some inspiration and exploration.

Results and Comparison:

In the following I will illustrate the difference between using imitation learning and using no imitation learning.

The problem when using no imitation learning is that the learning progress is dependend on random exploration and lucky guesses that lead to a better reward. This often results in plateaus during the training process that we have to break through with a good episode enrollment. However most of the time (especially when we use more complex data sets) the neural network is not able to explore the state space and learn how to finish the task. Therefore we have to guide the agent to the goal and show him how to finish correctly. This massively speeds up the whole training process.

Figure 1 shows the comparison between the finished tasks when using imitation learning and no imitation learning. With no imitation learning we reach a plateau at epoch 60 whereas with imitation learning the percentage of solved tasks is continously rising. After only 180 epochs we can solve 70% of the tasks with imitation learning but only 15% with no imitation learning.

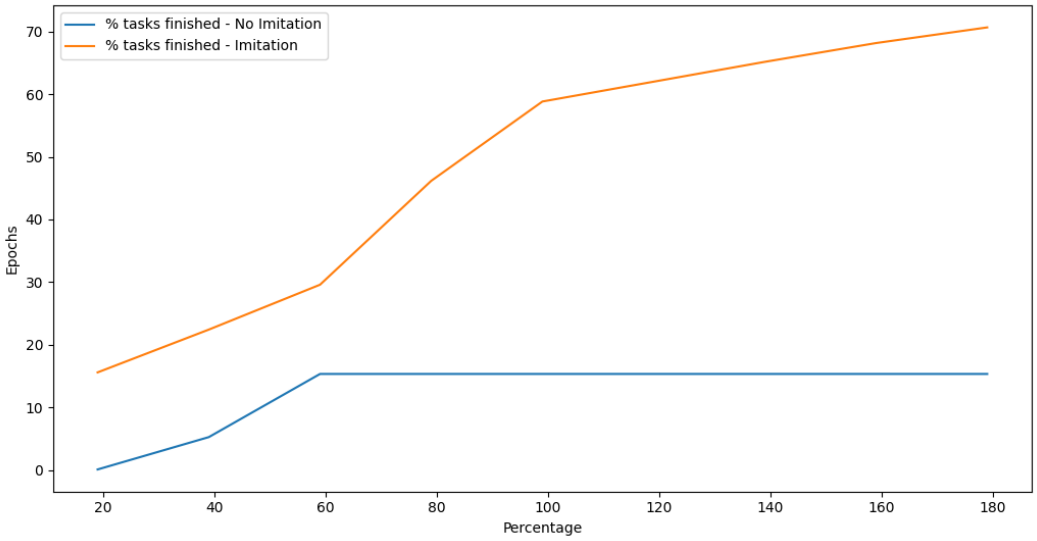

Figure 1: Depiction of the percentage of tasks solved when using imitation learning and no imitation learning on the medium data set.

Without imitation learning it is basically not possible to train the agent to solve tasks in the big data set.

(I.4)

Best performing configuration and general information:

- Model: CNN described above
- Optimizer: Adam
- Learning Rate: 0.0001
- Batch Size: 256
- Discount: 0.97
- Data Set: Big
- Imitation Learning: True
- Loss function for state value parameters: Smooth L1 Loss
- Complete Episode Enrollment
- max. Epochs: 3801

```
Model:
CNN(
        (relu): ReLU()
        (conv1): Conv2d(5, 7, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv2): Conv2d(7, 7, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (conv3): Conv2d(7, 7, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (value_l): Sequential(
                (0): Flatten(start_dim=1, end_dim=-1)
                (1): Linear(in_features=112, out_features=80, bias=True)
                (2): ReLU()
                (3): Linear(in_features=80, out_features=40, bias=True)
                (4): ReLU()
                (5): Linear(in_features=40, out_features=20, bias=True)
                (6): ReLU()
                (7): Linear(in_features=20, out_features=1, bias=True)
                (8): ReLU()
        )
        (policy_l): Sequential(
                (0): Flatten(start_dim=1, end_dim=-1)
                (1): Linear(in_features=112, out_features=80, bias=True)
                (2): ReLU()
                (3): Linear(in_features=80, out_features=40, bias=True)
                (4): ReLU()
                (5): Linear(in_features=40, out_features=20, bias=True)
                (6): ReLU()
                (7): Linear(in_features=20, out_features=6, bias=False)
        )
        (softmax): Softmax(dim=1)
)
```

The whole training time is: 21 hours, 11 minutes, and 51 seconds for 3081 epochs.

The figure 2 depicts the percentage of tasks solved and tasks solved with a minimal sequence of actions. We notice a steep improvement until ~ 500 epochs and a slow convergence to nearly 100%. After training for around 21 hours I still notices small improvements. If I would continue training for a much longer time I would reach even better results, I guess. But I am quite happy with a final percentage of approximately 97% tasks solved and 96.5% tasks solved optimally on the complete validation data set.

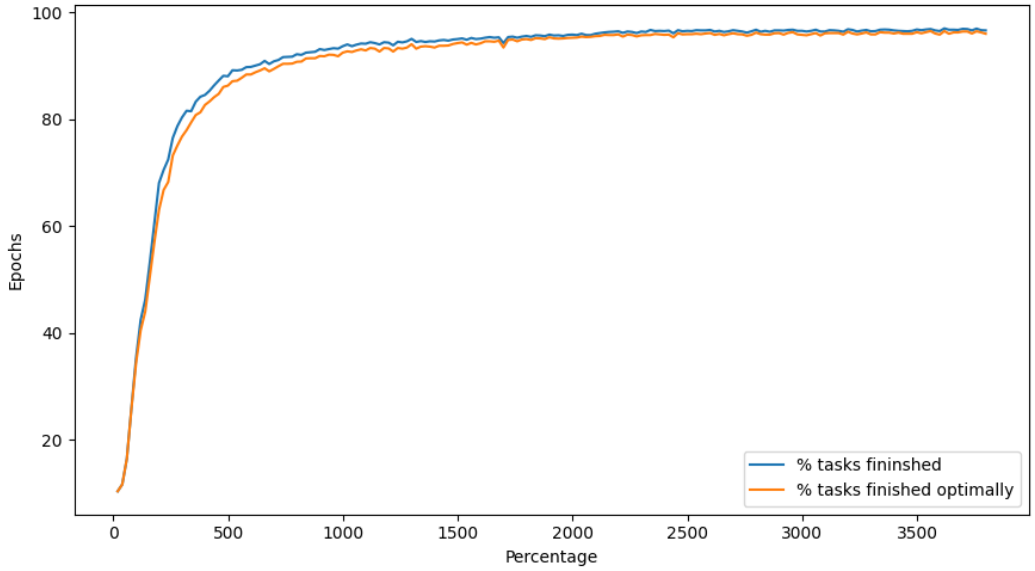

Figure 2: Depiction of the percentage of tasks solved and tasks solved with a minimal sequence of actions during the training of the network.

Figure 3 depicts the loss during the training process and the actions that were predicted correctly. This is a nice metric to directly have an idea of the performance of the network model during training without the need of running through the validation set during the training. Of course this is only possible with imitation learning.

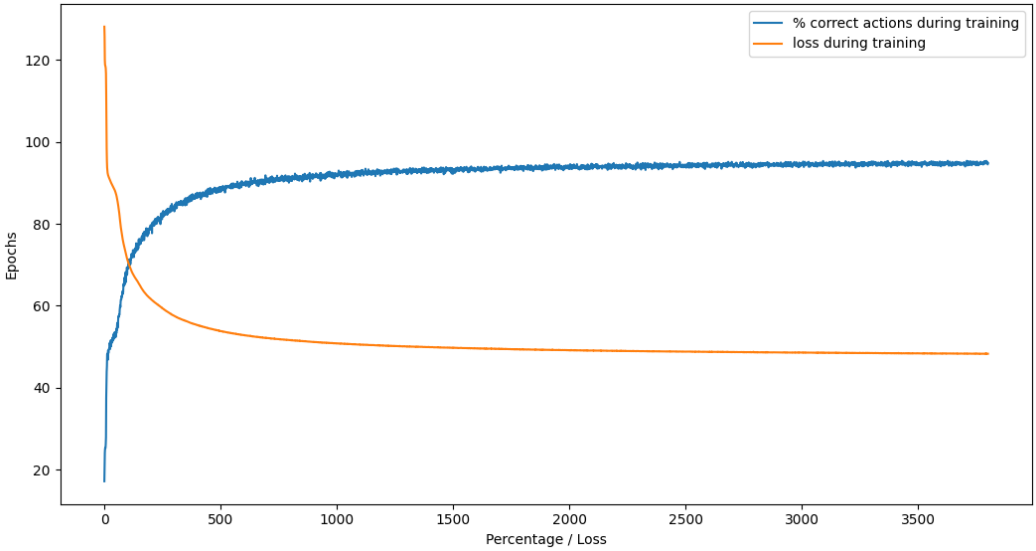

Figure 3: Depiction of the loss during the training and the correct predicted actions during training.

One epoch in the training of the final network model enrolls every task in the dataset. That means I enroll 24000 tasks with the optimal sequences. One epoch takes approximately 20 seconds. The backward pass needs around 67% of the time and the enrollment needs around 30% of the time. (3% is other stuff like logging and printing to console)

Finally the final neural network model solves 100% of the tasks of the medium and easy data set. For the medium data set it solves almost 100% (except one task) of the tasks with a minimal sequence. For the easy data set it solves 100% of the tasks optimally.