University of Alberta
Computing Science Department
Neuro-Symbolic Programming - W25

Assignment 3
15 Marks
Due date: 04/04 at 23:59

Overview

In this assignment, you will implement a masking scheme for extracting modules of trained feedforward neural networks with a single hidden layer of ReLU neurons. This assignment aims to answer the following question: *Given a neural network encoding multiple functions for solving a sequential decision-making problem, can we extract reusable sub-functions of the network with gradient descent?* Answering this question would allow us to obtain sub-functions that can more easily be applied to solve downstream problems.

The assignment provides an affirmative answer to this question. To answer the question, we will perform two tests. First, we will test if we can extract sub-networks from the original network that perform functions the original model performs as part of the solution it encodes. Second, we will show that we cannot extract functions that are not part of the solution the original network encodes. While it is easy to understand the need for the first test, the second test requires an explanation. A positive second test provides evidence that the result of the first test did not happen by chance.

You should submit a zip file with your implementation and a pdf with the answers to the questions.

Dependencies

You will need the following installed in your machine to run the starter code: torch, numpy, tyro==0.8.5, tabulate, gymnasium==0.29.1. All these dependencies can be installed by running

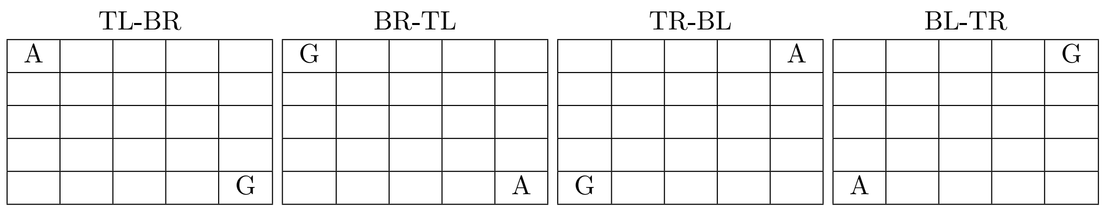
`pip install -r requirements.pip`.

Problem Definition

In this assignment, we will consider a simple sequential decision-making problem called ComboGrid. ComboGrid is a grid world where the agent must find a goal cell on a 5×5 grid. In contrast with other grid-world problems, where the agent can move up, down, left, and right, in ComboGrid the agent needs to perform sequences of actions (the combo) to attain the desired effect of moving up, down, left, and right. The actions available to the agent are 0, 1, and 2. Below we show each sequence and their effects.

- `0, 0, 1`: up
- `0, 1, 2`: down
- `2, 1, 0`: left
- `1, 0, 2`: right

The grids below present the four problems we will consider. Each problem is determined by the agent's location and the goal's location. The problem's name encodes their locations: TL-BR means top-left (agent) and bottom-right (goal), while BL-TR means bottom-left (agent) and top-right (goal).



To solve TL-BR, the agent needs to learn how to move down and right, to solve BR-TL up and left, to solve TR-BL down and left, and to solve BL-TR up and right. Note how each sequence is required in two problems. For example, the sequence `0, 0, 1` (up) must be used to solve BR-TL and BL-TR. This way, if an "up function" is extracted from a network encoding a solution to BR-TL, we could reuse it while trying to solve BL-TR. ComboGrid is a test bed for neural decomposition that is easy to understand and debug.

We offer one trained feedforward ReLU neural network for each of these problems. We will extract functions for each of the four sequences by masking the ReLU neurons of the hidden layer of the networks. This will be achieved by using the solution trajectories for each problem as training data. Then, we will collect each sequence's occurrences in the four trajectories, thus forming one training set for each sequence. See function `learn_masks` of `main.py`. The dictionary `sequences_to_learn` stores the training data for each sequence.

Given the data for one of the sequences, e.g., `0, 0, 1`, we will learn a mask that extracts a sub-function of a network that correctly maps each observation in the training data to its corresponding action. For example, in the training data for sequence `0, 0, 1`, we will have observations from the solution path of both BR-TL and BL-TR (the two problems that require the agent to move up). The goal is to learn a mask for each of the four neural networks given in the starter code, such that the masked network fits the training data.

Next, we will discuss how to learn the masks with gradient descent.

Masking Neurons

A ReLU neuron is given by the function $\max(0, Z)$, where Z is a linear function of the neuron's inputs. As we have seen in class, each ReLU neuron can be in one of two states: active and inactive. When the neuron is active, it outputs the Z function; when it is inactive, it outputs 0. A mask for a network will assign one of three values to each neuron: active, inactive, or "part of the extracted program". If a mask sets a neuron to be active, then, independently of its ReLU computation, that neuron will return Z; if a mask sets a neuron to be inactive, also independently of its ReLU computation, that neuron will always return 0. Finally, if the mask sets a neuron to be "part of the program", the neuron will return its ReLU value as usual: $\max(0, Z)$. A mask for a neural network determines the state of each neuron. For example, a mask for a network with three neurons would look like this: $[\text{Active}, \text{Active}, \text{Part of Program}]$, where the first two neurons are active and the third is part of the program. We use networks with 64 neurons, so the masks will have 64 entries.

In class, we showed the equivalence of a neural network and a neural tree. The number of possible sub-functions we can consider is equal to the number of sub-trees we can obtain from the tree. A mask determines exactly one of such sub-trees as follows. The mask of a neuron n being "active" means that we follow the branch of the node representing n in the tree that matches the rule $Z \geq 0$. If a neuron is inactive, we follow the branch that matches the rule $Z < 0$. Finally, if a neuron n is "part of the program", then the sub-tree we are considering as the extracted sub-function has the node representing n in it.

We will use a Softmax function to implement the masks of the neurons.[1] The Softmax transforms the

[1]This was a suggestion Daniel Zhang gave in class; it replaces the modified tanh approach by Koul, Fern, and Greydanus (2019) we discussed in class. We experimented with the tanh and Softmax approaches and found that the latter is more stable in training, so this is what we are using in the assignment. Thank you, Daniel!

parameters θ_i we will learn with gradient descent in a probability distribution that sums to one.

$$\text{Softmax}(\theta_i) = \frac{e^{\theta_i}}{\sum_{j=1}^{n} e^{\theta_j}}$$

We will have three θ-values for each neuron, for each possible value of the mask. Namely, θ_0, θ_1, and θ_2 are the parameters for inactive, active, and part of the problem, respectively. Once we run these parameters through a Softmax, we will obtain a probability distribution, from which we will take the maximum value. Recall that taking the maximum is a non-differentiable operation, and that is where we need to use the Straight-Through Estimator we have seen in class. Here is a complete example of these operations.

The following instruction creates a matrix of shape 3×64, such that each column contains the θ_0, θ_1, and θ_2 parameters for a single neuron. The value of `args.hidden_size` is 64 in the starter code.

```
mask = torch.nn.Parameter(torch.randn(3, args.hidden_size), requires_grad=True)
```

The Softmax for this matrix can be performed in Torch as follows, so we have a matrix also of shape 3×64 with one probability distribution in each column of the matrix.

```
mask_probs = torch.softmax(mask, dim=0)
```

Then, we perform the discretization step with the Straight-Through Estimator, which is given in `main.py`.

```
mask_discretized = STESoftmax.apply(mask_probs)
```

The discretized mask, a matrix of shape 3×64, whose columns contain a single one and two zeros, will be used to perform a masked forward pass in a neural network. The masked forward pass is something you will have to implement in the assignment (see it below). The forward pass can called as follows.

```
new_trajectory = agent.run_with_mask(observations, mask_discretized, trajectory.get_length())
```

Here, `agent` is an instance of the class `PPOAgent`. The method `run_with_mask` is given in the starter code. However, `run_with_mask` calls the method `_get_action_with_mask`, which should perform the masked forward pass. You will implement `_get_action_with_mask`. The variable `observations` is a list of observations collected from the training data of a given sequence (e.g., `0, 0, 1`). Let `trajectory` be the training data for a given sequence. The observations can be collected from the trajectory as follows.

```
observations = trajectory.get_state_sequence()
```

The variable `new_trajectory` is the output of the masked forward. That is the set of actions the model returns for the current masked model. We will then improve on the mask with gradient descent. To do so, you will use the Cross Entropy loss between the actions the from the training data, which can be obtained with `actions = trajectory.get_actions()` and the logits of `new_trajectory`. The code will look as follows.

```
loss_fn = torch.nn.CrossEntropyLoss()
mask_loss = loss_fn(torch.stack(new_trajectory.logits), torch.tensor(actions))
```

One iteration of gradient descent on this loss is given by the following.

```
optimizer.zero_grad()
mask_loss.backward()
optimizer.step()
```

Here, `optimizer` is the algorithm we use to optimize for the mask parameters. You can define it as follows.

```
optimizer = torch.optim.Adam([mask], lr=args.mask_learning_rate)
```

The optimizer is set to update only the parameters of the mask, and not the parameters of the agent (the agent is the neural model we have trained for you). We want to hold the agent fixed while updating the parameters of the mask. To ensure that the agent will not be updated, we can call the method `agent.eval()` at the beginning of the function to ensure that the gradients will not update the parameters of the agent. We are going to use the learning rate given in `main.py`, which can be accessed with `args.mask_learning_rate`.

Implement Masked Forward Pass (3 Marks)

You will implement the method `_get_action_with_mask` from `ppo_agent.py`. For your reference, a regular forward pass is done as follows, where `self.actor[j]` accesses the j-th operation in the network. You need to modify this function so that the computation of `hidden_relu` is masked.

```
    def _get_action(self, x_tensor):
        hidden_logits = self.actor[0](x_tensor)
        hidden_relu = self.actor[1](hidden_logits)
        logits = self.actor[2](hidden_relu)

        prob_actions = Categorical(logits=logits).probs
        a = torch.argmax(prob_actions).item()
        return a, logits
```

Implement Function for Learning Masks (7 Marks)

You will implement the function `train_masks` in `main.py`. First, you should read `main.py` to understand the context in which `train_masks` will operate. Once you understand the context, you will implement the function such that it closely follows the process described in the section "Masking Neurons" above. You will update the parameters of the mask for `args.mask_learning_steps` steps. You should print the loss as training progresses to determine if the optimization step is effectively improving the model. The loss should consistently decrease, but it is normal to observe some "bouncing around" during optimization.

Due to this "bouncing around" effect of the loss, we need to store in memory the mask that resulted in the lowest loss in the training data. That is the mask that this function will return, as opposed to the mask obtained after the last gradient update. In addition to the best mask encountered in training, it will also return the loss before optimization and the loss of the best mask. The return statement will be as follows.

```
return best_mask.detach().data, init_loss, best_loss
```

Evaluate the Learned Masks and Questions (5 Marks)

If you correctly implement the masked forward pass and the function for training masks, the starter code will automatically evaluate the trained masks on a set of test observations. Instead of attempting to use the masked models (i.e., extracted functions) in downstream tasks, we will evaluate whether the functions generalize to observations not seen in training. This will be done by running the masked model on all cells of the four problems, TL-BR, BR-TL, TR-BL, and BL-TR, excluding the goal location.

The starter code will then generate a set of tables reporting on the percentage of the test observations in which the masked model and the original model, correctly executed the actions of the target sequence.

The table below shows the result for the up sequence (`0, 0, 1`). Here is how to read this table. In the first row, the masked model trained on the neural network that can solve the TL-BR problem (Base Model) correctly mapped the sequence actions in 68.7% of the observations from the test data; the original model, with no masking, correctly mapped 33.33% of the observations.

```
+------------+----------------+------------------+--------------+
| Sequence   |   Masked Score |   Original Score | Base Model   |
+============+================+==================+==============+
| Up         |       0.6875   |         0.333333 | TL-BR        |
+------------+----------------+------------------+--------------+
| Up         |       0.621528 |         0.204861 | TR-BL        |
+------------+----------------+------------------+--------------+
| Up         |       0.96875  |         0.583333 | BR-TL        |
+------------+----------------+------------------+--------------+
| Up         |       0.8125   |         0.631944 | BL-TR        |
+------------+----------------+------------------+--------------+
```

Considering the tables you obtained, answer the following questions.

1. (3 Marks) Discuss the results you have obtained considering the two tests that we wanted to run for this assignment. That is, how do the percentage values of learned masks for a sequence s compare for base models trained to solve problems that require s with base models trained to solve problems that do not require s? Explain the results you observe.

2. (2 Marks) What are we missing in our masking-learning scheme so that we can potentially obtain better percentage values than those you observed in your table of results? In this question, I would like you to consider the scenario in which the optimization process of finding the mask parameters can be solved exactly. That is, I would like you to think of reasons to increase the percentage values that do not involve blaming the optimization process.