

Background Information For Project

Introduction to Deep Reinforcement Learning with Formal Verification

Written By: Shaiel Vistuch

Supervisors: Avraham Raviv, Prof. Hillel Kugler

Main chapters

Verification of Deep Neural Networks (and DRL)	2
Reinforcement Learning	5
Deep Reinforcement Learning	10
Propositional Logic.....	17
Introduction to NuSMV.....	18
NuSMV Basic Commands:.....	20
Verification of Vanila Q-learning.....	21
The Marabou verification tool	22

Extended Table Of Content

Verification of Deep Neural Networks (and DRL)	2
Activation functions.....	3
Combining DNNs with formal verification	3
Theory Solver	3
The Reluplex algorithm.....	4
k-induction verification approach.....	5
Invariant verification approach	5
Resources for section.....	5
Reinforcement Learning	5
Basic Terminology:.....	5
How to evaluate the Q-value?	6
Temporal Difference (TD) method	6
Temporal Difference n -return method	7
Temporal Difference λ -return method.....	7
Backwards view of Temporal Difference λ -return method.....	8
Q-learning method	9
Exploration vs exploitation	9
Resources for section.....	10
Deep Reinforcement Learning	10
Batch vs Incremental learning	10

Possible Python libraries.....	11
Building a network in PyTorch	11
Defining the edges of layers in PyTorch	11
Defining activation functions.....	12
Deep Q-Network.....	13
Fixed Q-targets technique	13
Replay buffer technique	14
Gradient Descent	14
Vanilla Deep Q-learning algorithm	14
Standard Deep Q-learning algorithm.....	14
Double Deep Q network (DDQN).....	16
Policy based, value- based, and Actor-Critic algorithms	16
Actor-Critic policy gradient algorithm	16
Resources for section.....	17
Propositional Logic.....	17
Common Symbols:	18
Rules for natural deduction:	18
Resources For section:	18
Introduction to NuSMV.....	18
Hands-on example:	19
NuSMV Basic Commands:	20
Resource for sections:	21
Verification of Vanila Q-learning.....	21
The Marabou verification tool	22
Defining a network	22
Making a query	24
Making a query using the command line	25
Making a query using the python interface.....	25
Disjunctions	26
Resources for section.....	26

Verification of Deep Neural Networks (and DRL)

A neural network is comprised of a set of layers of nodes, each node is called a neuron (named after the neuron in the human's brain). Each neural network has an input layer, at least one "hidden layer", and an output layer.

The neuron is the most basic computational unit of any neural network. To put it simply, a neuron takes an input, process it, and pass a value to other neurons present in the multiple layers of the network. More formally speaking, the value in each neuron is determined by computing a linear combination of values from nodes in the preceding layer and then applying an *activation function* to the result, then it passes the value to the neurons in the next layer and this cycle continues until the processed data reaches the output layer.

Activation functions

In the human brain, when receiving information, we classify the information to what's useful and what's not useful to us. Activation functions are functions that helps the neural network sort the "useful" information from the "not-useful" information.

Linear activation functions could be used to solve a very limited set of cases. Usually, these types of networks will generate a linear combination of inputs which can also be done in just one step. In other words, it behaves like a single layer network. In this section we will want to look at Deep Neural Networks (DNNs), which have more than a single hidden layer. In these networks, the activation functions can be non-linear.

There are many different types of activation functions, we will usually pick the type based on the problem specification. In this section we are going to focus a specific kind of activation function, called a Rectified Linear Unit (ReLU).

When an ReLU function is applied to a node there are two cases: if the node has a positive value, it returns the value unchanged (this is called the active case), if the value is negative, the ReLU function returns 0 (this is called the inactive case).

Combining DNNs with formal verification

By training DNNs, we can replace creation of difficult software. The performance of these DNNs is often comparable to, or even surpasses, the performance of manually crafted software.

As artificial intelligence takes a more important part in our world, we must develop methods that can provide formal guarantees about a DNN behavior. This is what we are going to try to do in this section, for DNNs with ReLU.

Theory Solver

A *theory* is a pair $T = (\Sigma, I)$ where:

- Σ is a signature
- I is a class of Σ -interpretation

Σ -formula is *T-satisfiable* if it is satisfied by some interpretation in I .

A *quantifier-free* formula is a formula that does not contain the symbols \exists or \forall . Instead, it can be made of the Boolean operators \wedge (and), \vee (or), and \Rightarrow (implies), equations and inequalities. We are only going to consider quantifier-free formulas. (See [2]) .

$T_{\mathbb{R}}$ is made of the signature containing all the constant rational numbers, and the symbols $=, \leq, \geq, \cdot, +, -$, paired with the standard model of the real numbers. In this section we would focus on linear formulas, meaning that we are only going to focus on formulas over $T_{\mathbb{R}}$ in which the multiplication symbol is applied only if one of the factors is a rational constant.

Linear atoms can always be rewritten into the form $\sum_{x_i \in X} c_i x_i \bowtie d$ where:

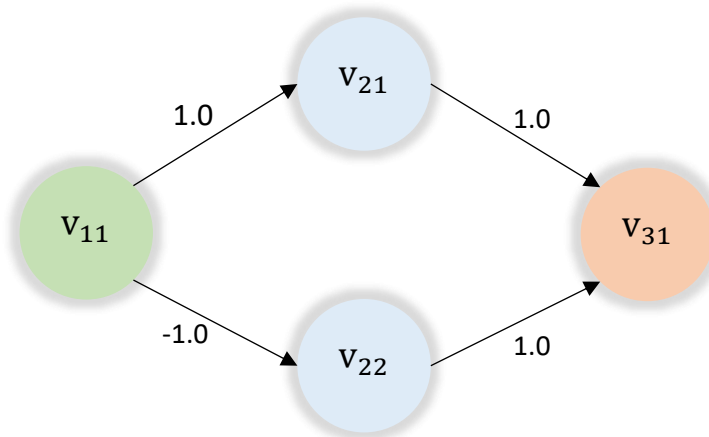
- \bowtie is one of the algebraic operators $=, \leq, \geq$.
- X is a set of variables.
- c_i, d are irrational constants.

The Reluplex algorithm

The simplex algorithm is used to determine the $T_{\mathbb{R}}$ -satisfiability of conjunctions ($=$ combination, \wedge) of linear atoms. Unfortunately, nonlinear ReLUs cannot be encoded this way. This is why a new algorithm – Reluplex – was created [1].

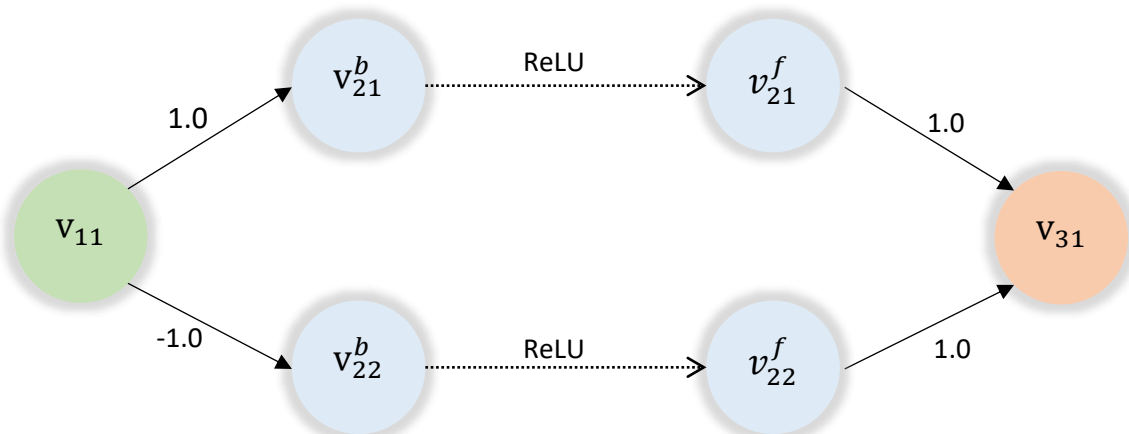
For the Reluplex algorithm we will define a new theory $T_{\mathbb{R}R}$. $T_{\mathbb{R}R}$ is identical to $T_{\mathbb{R}}$ except that its signature Σ includes also the binary predicate ReLU with the interpretation: $\text{ReLU}(x, y) \Leftrightarrow y = \max(0, x)$. Therefore, the formulas that can be formed over $T_{\mathbb{R}R}$ are made of atoms that are either linear inequalities or applications of the ReLU predicate to linear terms. So we can easily encode DNNs and their linear properties as conjunctions of $T_{\mathbb{R}R}$ atoms.

Let's look at a network with a hidden layer that uses an ReLU activation function:



The idea of the Reluplex algorithm is to “split” each ReLU node v into a pair of variables, v^b, v^f and then assert $\text{ReLU}(v^b, v^f)$ (remember that by definition $\text{ReLU}(v^b, v^f) \Leftrightarrow v^f = \max(0, v^b)$).

- v^b is the backward facing variable, used to connect to the preceding layer.
- v^f is the forward-facing part of the variable, used to connect to the next layer.



Full explanation of the algorithm is described in pages 8-10 in the paper [1].

k-induction verification approach

The k-induction verification technique works with a constant k that represents the number of steps that we can take during our search for a proof or a counter example.

The idea is that first we are going to check if the desired property holds for $k=1$, and if we still didn't find a counter example, we will search for it with $k=2$, and increase k like so until we find out for sure if the property holds. This technique is supposed to save resources by hoping that we can prove/disprove the desired property in a limited number of steps. By doing so, we can save time and resources.

Invariant verification approach

An invariant is a partition of the state space S into two disjoint sets, S_1 and S_2 , such that no transition leads from one set to the other. We can verify if a property holds if we find an invariant such that all initial states are in S_1 and all the 'bad' states are in S_2 .

Resources for section

[1] Guy Katz, Clark Barrett, David Dill, Kyle Julian, Mykel Kochenderfer,
<https://arxiv.org/abs/1702.01135>

[2] 'quantifier', in planetmath
[https://planetmath.org/quantifier#:~:text=A%20quantifier%20is%20a%20logical,and%20%E2%88%83%20\(there%20exists\)](https://planetmath.org/quantifier#:~:text=A%20quantifier%20is%20a%20logical,and%20%E2%88%83%20(there%20exists))

[3] Guy Amir, Michael Schapira and Guy Katz, *Towards Scalable Verification of Deep Reinforcement Learning*, 2021

[4] Yafim Kazak, Clark Barrett, Guy Katz, Michael Schapira, *Verifying Deep-RL-Driven Systems*

[5] *Activation Functions in Neural Networks [12 Types & Use Cases]* by Pragati Baheti
<https://www.v7labs.com/blog/neural-networks-activation-functions>

Reinforcement Learning

Basic Terminology:

Agent: an entity that operates within an environment.

States: a variable that describes the current position of the agent.

Environment: The environment includes all the possible states for the agent.

Actions: They are the agent's possible operations when it is in a specific state.

Episodes: An episode is a collection of actions taken by the agent until it can no longer take a new action or the programmer decided to mechanically end it.

Q-values: The Q-value is the metric used to measure an action at a particular state. The higher the Q-value, the better it is to take the action.

Rewards: For each action the agent can take, a fitting reward is provided. This allows the agent to get feedback and learn if he is in the right direction to solving its task.

Reinforcement learning models use trial-and-error experiences to learn how to solve a problem in an optimal matter.

	Markov process	Markov Reward (Ex: DP, MC, TD)	Markov Decision Process (Ex: Q-learning)
S – set of states	✓	✓	✓
P - state transition probability matrix	✓	✓	✓
R - reward function		✓	✓
γ is a discount factor, $\gamma \in [0, 1]$		✓	✓
A - a finite set of actions			✓

How to evaluate the Q-value?

Q-Learning is an off-policy algorithm based on the TD method. We will first learn about TD and then go back to Q-learning.

Temporal Difference (TD) method

The temporal difference methods are a combination of two concepts: *Deep Programming* (DP) and *Monte Carlo* (MC). Detailed explanation for each of these two concepts can be found in David Silver's RL lectures 4 and 5, and in this [2] Medium article series.

The main differences between *Deep Programming* and *Monte Carlo*, is that Monte Carlo methods use sampling from the environment for updating values.

Dynamic Programming (DP) methods use the current estimations of values to carry out updates. This is called bootstrapping.

The temporal difference formula combines these concepts. Temporal difference methods have an additional parameter named λ , which we will talk about later. For now, let's see how DP and MC concepts are combined in a basic temporal difference method, named $TD(0)$:

$$\underbrace{V(s)}_{\substack{\text{New value} \\ \text{Estimation}}} = \underbrace{V(s)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} + \alpha * \overbrace{\left(\underbrace{r}_{\substack{\text{immediate} \\ \text{reward}}} + \gamma * \underbrace{V(s')}_{\substack{\text{Sample of the} \\ \text{next state} \\ \text{after } s}} - \underbrace{V(s)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} \right)}^{\substack{\text{TD error} \\ \text{TD target value}}}$$

Where:

α is the learning rate

γ is the discount factor

By adding the reward r , we sample the environment to update the value, like in *Monte Carlo*.

By using $V(s', a')$, we sample the immediate next state value to update the state, like in *Deep Programming*.

We can also write the formula using notations:

$$\underbrace{V(S_t)}_{\substack{\text{New value} \\ \text{Estimation}}} = \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} + \alpha * \left(\overbrace{\underbrace{r_{t+1}}_{\substack{\text{immediate} \\ \text{reward}}} + \gamma * \underbrace{V(S_{t+1})}_{\substack{\text{Sample of the} \\ \text{next state} \\ \text{after } s}}}_{\substack{\text{TD target value} \\ \text{TD error}}} - \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} \right)$$

The TD target value is sometimes called estimated return and marked

$$G_t = \underbrace{r_{t+1}}_{\substack{\text{immediate} \\ \text{reward}}} + \gamma * \underbrace{V(S_{t+1})}_{\substack{\text{Sample of the} \\ \text{next state} \\ \text{after } s}}$$

So we can write the equation as:

$$\underbrace{V(S_t)}_{\substack{\text{New value} \\ \text{Estimation}}} = \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} + \alpha * \left(\overbrace{\hat{G}_t}^{\substack{\text{TD target value} \\ \text{TD error}}} - \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} \right)$$

Temporal Difference n -return method

Let's look at the estimated return for n -steps:

$$\begin{aligned} G_t^{(1)} &= r_{t+1} + \gamma * V(S_{t+1}) \\ G_t^{(2)} &= r_{t+1} + \gamma * r_{t+2} + \gamma^2 * V(S_{t+2}) \\ &\vdots \\ G_t^{(n)} &= r_{t+1} + \gamma * r_{t+2} + \dots + \gamma^{n-1} * r_{t+n} + \gamma^n * V(S_{t+n}) \end{aligned}$$

In the n -return method, an additional parameter, n , is used. In short, we call this method $TD(n)$. We define the n -return method like so:

$$\underbrace{V(S_t)}_{\substack{\text{New value} \\ \text{Estimation}}} = \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} + \alpha * \left(\overbrace{\hat{G}_t^{(n)}}^{\substack{\text{TD target value} \\ \text{TD error}}} - \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} \right)$$

Temporal Difference λ -return method

In the TD λ -return method, an additional parameter, λ , is used. The n parameter is also used. In short, we call this method $TD(\lambda)$.

We define: $G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$

And the step in TD(λ) is defined to be:

$$\underbrace{V(S_t)}_{\substack{\text{New value} \\ \text{Estimation}}} = \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} + \alpha * \left(\overbrace{\widetilde{G}_t^\lambda}^{\substack{\text{TD error} \\ \text{TD target value}}} - \underbrace{V(S_t)}_{\substack{\text{Former} \\ \text{value} \\ \text{Estimation}}} \right)$$

Backwards view of Temporal Difference λ -return method

We're now going to take a different approach to TD(λ). This approach is called *backward view*.

Remember that G_t^λ depends on $V(S_{t+n})$, and that we use G_t^λ to update $V(S_t)$.

In the backward view approach, we use the current and past to update the state-values for every state we've seen so far, as opposed to updating using a future event.

We can establish this idea using eligibility traces. An eligibility trace is a record of the recent states and actions that an agent has visited. It is often represented as a matrix with the same dimensions as the Q-value table, where each element corresponds to a specific state-action pair.

In the TD(λ) algorithm, the eligibility trace is used to update the values of Q-function. It allows the algorithm to credit not only the most recent action but also past actions that contributed to the current outcome.

An eligibility trace is defined by:

$$E_0(s) = 0$$

$$E_t(s) = \gamma \lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$

where:

$\mathbf{1}$ is an indicator function, equals to 1 when the condition inside it is true and 0 otherwise.

This eligibility trace will be used as a scaling factor for the TD error. So, for example, if we're interacting with the environment and get a reward r_{t+1} , our update to the state-value function will be:

$$V(s) = V(s) + \alpha * \delta_t * E_t(s)$$

$$\text{where: } \delta_t = r_{t+1} + \gamma * V(S_{t+1}) - V(S_t)$$

Notice that when $\lambda = 0$, only the current state is updated, since:

$$E_t(s) = \underbrace{\gamma \lambda E_{t-1}(s)}_0 + \mathbf{1}(S_t = s) = \mathbf{1}(S_t = s)$$

And we get:

$$V(s) = V(s) + \alpha * \delta_t * \mathbf{1}(S_t = s)$$

Substituting for δ_t we get:

$$V(s) = V(s) + \alpha * (r_{t+1} + \gamma * V(S_{t+1}) - V(S_t)) * \mathbf{1}(S_t = s)$$

Which means:

$$V(S_t) = V(S_t) + \alpha * (r_{t+1} + \gamma * V(S_{t+1}) - V(S_t))$$

And $V(s) = V(s)$ for $S_t \neq s$, just like in the TD(0) method we wrote before!

Notice that TD(1) total update is exactly the same as MC.

For further reading see: <https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html>

Q-learning method

In q-learning, we consider off-policy learning of action-values pairs marked as $Q(s,a)$.

Bellman's equation is used to help calculate the value of a given state and assess its relative position. The state with the maximum value is considered the optimal state. The formula is:

$$\underbrace{Q(s,a)}_{\substack{\text{New } Q\text{-value} \\ \text{Estimation}}} = \underbrace{Q(s,a)}_{\substack{\text{Former} \\ Q\text{-value} \\ \text{Estimation}}} + \alpha * \overbrace{\left(\underbrace{r}_{\substack{\text{immediate reward} \\ \text{for taking} \\ \text{action } a \text{ at state } s}} + \gamma * \underbrace{\max(Q(s',a'))}_{\substack{\text{Estimated optimal} \\ Q\text{-value of the} \\ \text{next state}}} - \underbrace{Q(s,a)}_{\substack{\text{Former} \\ Q\text{-value} \\ \text{Estimation}}} \right)}^{Q\text{-learning error}}$$

$$\underbrace{Q(s,a)}_{\substack{\text{New } Q\text{-value} \\ \text{Estimation}}} = \underbrace{Q(s,a)}_{\substack{\text{Former} \\ Q\text{-value} \\ \text{Estimation}}} + \alpha * \left(\underbrace{r}_{\substack{\text{immediate reward} \\ \text{for taking} \\ \text{action } a \text{ at state } s}} + \gamma * \underbrace{\max(Q(s',a'))}_{\substack{\text{Estimated optimal} \\ Q\text{-value of the} \\ \text{next state}}} - \underbrace{Q(s,a)}_{\substack{\text{Former} \\ Q\text{-value} \\ \text{Estimation}}} \right)$$

$$Q(s,a) = Q(s,a) + \alpha * (r + \gamma * \max(Q(s',a')) - Q(s,a))$$

In the code, we do this using:

```
# Updating qtable
self.qtable[state, action] = self.qtable[state, action] +
self.learning_rate * (
    reward + self.gamma * np.max(self.qtable[new_state, :]) -
self.qtable[state, action])
```

Exploration vs exploitation

With a probability of $1 - \epsilon$, we do exploitation, and with the probability ϵ , we do exploration.

In the epsilon – greedy policy we will:

- Generate the random number between 0 to 1.
- If the random number is greater than epsilon, we will do exploitation. It means that the agent will take the action with the highest value given a state.
- Else, we will do exploration (take a random action).

In the code, I wrote an epsilon-greedy function to pick the next action:

```
def epsilon_greedy(self, state):
```

```

random_num = random.uniform(0, 1)

# Exploitation: picking max Q value
if random_num > self.epsilon:
    return np.argmax(self.qtable[state, :])

# Exploration: picking a random action
else:
    return self.env.get_random_action().value

```

As training progresses, we need less exploration and more exploitation. We can decrease epsilon using Linear Decay, Exponential Decay, or Discrete Interval Decay methods.

In the code, we are doing exponential decay using:

```

self.epsilon = (self.max_epsilon - self.min_epsilon) * np.exp(-
self.decay_rate * episode) + self.min_epsilon

```

Resources for section

1. David Silver's DeepMind RL course
<https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9->
2. Dan Lee, *Reinforcement Learning, Part 5: Monte-Carlo and Temporal-Difference Learning*,
<https://medium.com/ai%C2%B3-theory-practice-business/reinforcement-learning-part-5-monte-carlo-and-temporal-difference-learning-889053aba07d>
3. *Q-learning*, TechTarget, <https://www.techtarget.com/searchenterpriseai/definition/Q-learning>
4. *Reinforcement Learning: Eligibility Traces and TD(λ)*, Alister Reis's blog
<https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html>

Deep Reinforcement Learning

So far we have represented value function by a lookup table – either by having each state have a matching value $V(s)$ or by having each action-value pair with value $Q(s, a)$.

There are many disadvantages to that: for a large set of states and/or actions, the lookup table would consume a lot of memory. In addition, in the lookup-table methods, we must learn the right value for each state individually, which slows down the learning process.

A solution for that, is that instead of getting our values from a table, we would get it from a function. These types of functions are called *value function approximators*. We can define a state using the function inputs. The output of the value of the function would be the value(s) of the state we have defined. There are many ways of implementing those value function approximators, but we are going to focus on one: a neural network.

Batch vs Incremental learning

In Batch Learning methods, we take a set of data, and train our network using this complete set of data. A possible method for Batch methods to store the data for late training is by using a *replay buffer*.

Incremental learning is used when a full training set of data isn't available. In incremental learning you must train and update the network at each step you take, you can't simulate steps and save the results for later training.

Possible Python libraries

Tensorflow, keras, PyTorch.

We will use PyTorch.

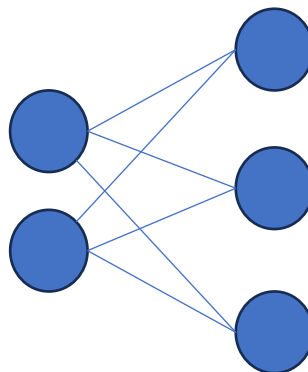
Building a network in PyTorch

We can build a network data structure by defining a DQN class. Our class would inherit from `torch.nn.Module`, which is the PyTorch base class for all neural network modules. Usually, we import `torch.nn` as `nn`.

Each network in PyTorch must have two methods: an `__init__` method, that sets up the network's structure by defining the layers of edges and their weights, as well as all the layers that have learnable parameters that are going to change during training, and a method named `forward`, which specifies how the data is changed in each layer of nodes. The `forward` method is responsible for taking data as input and passing it through the layers of the network to produce the output.

If only the weights of the network's edges are changed during the training, we can say that the `__init__` method defines the layers of edges while the `forward` method defines the activation functions (i.e. the layers of nodes).

There are many ways to create layers of edges. But the most common type of an edge layer is called a linear layer. In this layer, each node in the preceding node layer is connected to each node in the next node layer. Linear layers are a type of fully connected layers.



An example of a Fully Connected Network

A *fully connected* network is a network in which none of the weight matrices have zero entries. Each node receives a value, processes it using an activation function, and outputs the result. In linear layers of edges, the value that the node receives as an input is a linear combination of all the inputs of the previous layer of nodes.

Defining the edges of layers in PyTorch

To define a layer, we use:

```
torch.nn. + layerType (such as conv1d, relu, linear)
```

We define the layers with learnable parameters in the `__init__` function.

For example: `self.layer1 = nn.Linear(n_observations, 128)` would create a linear layer. That is, `self.layer1` would be of class `torch.nn.modules.linear.Linear`.

Defining activation functions

`torch.nn.functional` is used to carry out functions. It is usually imported as `F` to make it shorter to write. For example, `x = F.relu(self.layer1(x))` gets inputs and performs a linear transformation to them to obtain the outputs.

In conclusion, `torch.nn` and `torch.nn.functional` can be used to with common functions such as: `conv1d`, `relu`, `linear`.

Q: What's the difference between `torch.nn.functional.linear` and `torch.nn.linear`? More generally speaking, what's the difference between putting functions names (such as `conv1d`, `relu`, `linear`) after `torch.nn` vs after `torch.nn.functional`?

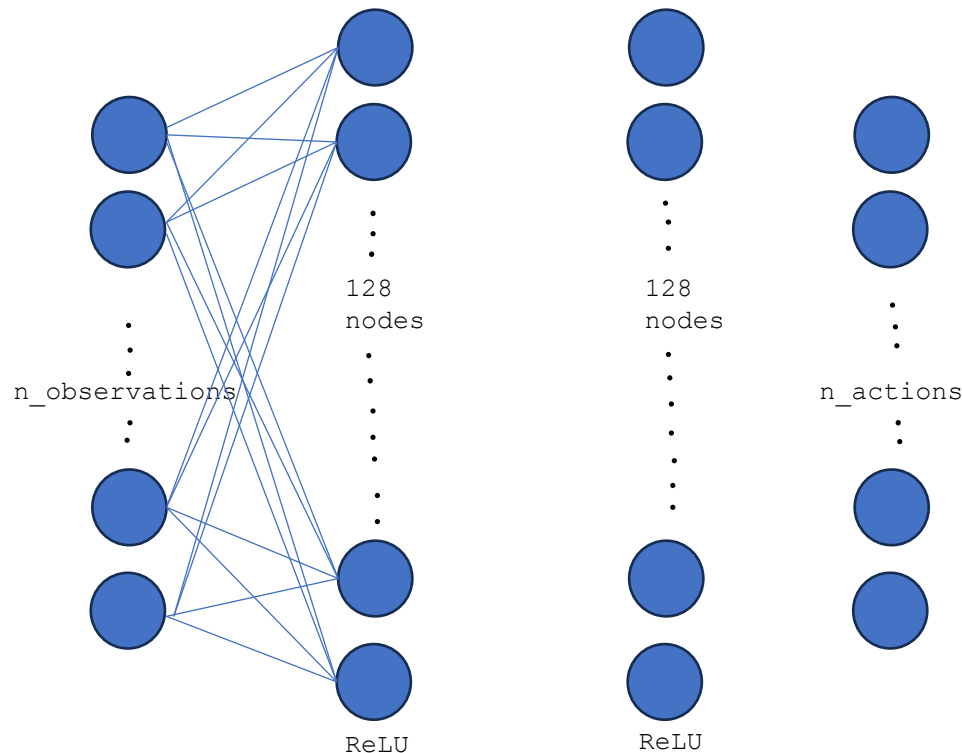
A: Well, `torch.nn.functional.linear` is an operator that calculates what happens to a data after passing through the given layer. On the other hand, `nn.linear` defines the weights and parameters of the edges that the network learns during training. Initially, they are set to random values.

```
import torch.nn as nn
import torch.nn.functional as F

class DQN(nn.Module):

    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.layer1 = nn.Linear(n_observations, 128)
        self.layer2 = nn.Linear(128, 128)
        self.layer3 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        return self.layer3(x)
```



Deep Q-Network

I have already discussed Deep Networks in a previous section. In the context of DQN-learning, our input layer would receive the current state, and the output layer would output the value of taking each possible action at the current state. The action with the maximum Q-value is selected from the main Q-network.

The idea is, that instead of updating the values themselves, we would update the way we obtain the values. We do that by making changes to the network itself. This is done by changing the weights of the network's edges. (There are also some activation functions with learnable parameters, but I won't be using them here).

Fixed Q-targets technique

The fixed Q-targets technique involves the use of two separate networks. The purpose of having two networks is to prevent over-estimation. The two networks are commonly referred to as the main Q-network and the target Q-network.

Main Q-Network:

This network is the primary network responsible for estimating Q-values. This would be our final network which we will use after we are done with the training process. During each iteration of training, this network is updated.

Target Q-Network:

This network is used only during the training process. Its weights are not updated as frequently as the main Q-network. Instead, they are periodically updated with the weights of the main Q-network. The Q-learning target value is the output of the Target Network plus the reward from the sample.

$$\overbrace{r + \gamma * \max(Q_T(s', a'))}^{\text{Deep Q-learning target value}}$$

immediate reward for taking action a at state s *Estimated optimal Q_T-value of the next state*

TIP: You can remember that the target value is calculated from the target network.

Replay buffer technique

A replay buffer is a memory used to store experiences (state, action, reward, next state) during interactions with the environment. Instead of updating the Q-network after each interaction, we would store a batch of experiences and sample it once in a while during the training process. Only the sampled experience would affect the network. The purpose of this technique is to prevent over-training of the network.

Gradient Descent

The gradient is the vector of partial derivatives:

$$\nabla f(x_1, x_2, x_3 \dots x_n) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots \frac{\partial f}{\partial x_n} \right)$$

In order to the critical points of f , we need to solve the system of equations: $\nabla f = 0$.

When working with gradient descent, we are interested in the direction of the fastest decrease in the cost function. This direction is determined by the negative gradient, $-\nabla f$.

Once you have a random starting point $w = (w_1, \dots, w_n)$, you update it, or move it to a new position in the direction of the negative gradient: $w \rightarrow w - \eta \nabla f$, where η is a small positive value called the learning rate.

For further reading see: <https://realpython.com/gradient-descent-algorithm-python/>

Vanilla Deep Q-learning algorithm

In this algorithm, we perform a similar process to the vanilla Q-learning algorithm, only that instead of a table, we use a network for estimations. It's similar to performing the standard deep Q-learning algorithm when $N=1$.

Standard Deep Q-learning algorithm

In David Silver's lecture 6 slide 36, he explains the deep Q-learning algorithm with both fixed Q-targets and replay memory buffer. The algorithm works like so:

- 1) Initialize replay memory D . Initialize a deep main Q-network with random weights, Q_M .
Initilize a second network, called target network, to have the exact same values as the main Q-network, Q_T .
- 2) For $ep_count < total_episodes$:
 - a. For $step_count < max_steps$:
 - i. Do N times:
 1. Simulate an action a_t according to epsilon-greedy policy. If $\epsilon > random_number$, we would simulate the action that maximize the

value according to the Main Q-network. Otherwise, we would simulate a random action.

2. Store the transition in replay memory: that is, the current state s_t , the action a_t , the reward r_{t+1} , and the next state s_{t+1} .
Note: We don't change the values of the networks! We simulate T steps according to the current Main Q-network policy.
3. Sample a random mini-batch of transitions (s, a, r, s') from the replay memory and use them to update the main Q-network:
4. Use the target Q-network to calculate the target Q-values for the chosen actions in the minibatch. The target Q-values are computed as the sum of the immediate reward and the discounted maximum Q-value of the next state.

$$\overbrace{r + \gamma * \max(Q_T(s', a'; w_{T,i}))}^{\text{Deep Q-learning target value}}$$

immediate reward for taking action a at state s *Estimated optimal Q_T-value of the next state*

Where $w_{T,i}$ is the weight vector of the target network.

5. Minimize the Mean Squared Error (MSE) between the predicted Q-values from the main Q-network and the target Q-values, using a variation of stochastic gradient descent with respect to the weight vector of the Main Q network. So we need to minimize the loss function:

$$l_i(w_i) = \left(\overbrace{(r + \gamma * \max(Q_T(s', a'; w_{T,i})))}^{\text{Deep Q-learning target value}} - Q_M(s, a; w_{M,i}) \right)^2$$

Where:

$w_{T,i}$ is the weight vector of the Target network at the i -th iteration.
 $w_{M,i}$ is the weight vector of the Main network at the i -th iteration.
 In other words, the purpose is to find a new set of weights $w_{M,i}$ that minimize the loss function, and this is usually done using the gradient descent technique.

Note: if state s' is a terminal state (that is, the episode comes to an end after reaching state s'), then our loss function would be instead:

$$l_i(w_i) = (r - Q_M(s, a; w_{M,i}))^2$$

6. step_count = step_count + 1
 - ii. Update Target Network to be the same as the main Q-network.
- b. ep_count = ep_count + 1

NOTE: In the above algorithm we assume that the replay memory is large enough to hold how much transitions as we want. In a practical view, we would want to minimize this memory, and we will need to add some sort of method in which we replace irrelevant transitions with new ones (for

example, keeping a TTL on transitions stored in memory or replacing the oldest transitions each time).

Double Deep Q network (DDQN)

The target network in the DQN architecture provides a natural candidate for being a second value function, without having to introduce additional networks.

The idea is, that the target Q-learning value becomes:

$$\overbrace{r + \gamma * Q_T(s', a'; w_{T,i})}^{\text{Deep Q-learning target value}}$$

where: $a' = \text{action that } \max(Q_M(s', a'; w_{N,i}))$

So basically, an action is chosen using the main Q-network and then the Q value is calculated using that action and the target network.

Policy based, value- based, and Actor-Critic algorithms

In **Policy-based** methods we explicitly build and save a representation of a policy that maps a state to an action. (For example: Monte-Carlo policy gradient method from lecture 7 of David Silver's course).

In **Value-based** we don't build a policy explicitly. The policy is to pick the action with the best value, and the value is derived from a value function. For example, DQN is a value-based algorithm, since it learns a value function, which maps a state to Q-values. At each steps we pick the Q-value with the maximum value.

Actor-critic is a mix of the policy-based and value-based methods (though it is sometimes regarded as policy-based). This means that it learns a policy, which **maps states to actions**. The policy represents the agent's best guess of what action to take in a state.

Actor-critic reinforcement learning uses two neural networks: a policy network and a value network. The policy network learns the policy, and the value network learns the value function.

Actor-critic methods maintain two sets of parameters, w and θ .

The critic is used to estimate the action-value function and update the action value parameters w .

The actor is used to update the policy parameters θ , in direction suggested by critic.

In conclusion, the critic is used to measure how good an action is. The actor takes the actions accordingly.

Actor-Critic policy gradient algorithm

INPUTS:

- a differentiable policy parameterization $\pi(a|s, \theta)$ that given parameters θ and state s it outputs an action a .
- a differentiable action-state value function parameterization $V(Q(s, a_1), Q(s, a_2) \dots Q(s, a_n)|s, w)$ which returns a value given a state s and parameters w . In other words, we need a value function approximator.

- Step size parameters: step size $\alpha > 0$ for updating the θ parameters and step size $\beta > 0$ for updating the w parameters.
- NOTE: θ, w are usually vectors

Algorithm:

- 1) Initialize the policy parameters $\theta \in \mathbb{R}^{d'}$ and weights $w \in \mathbb{R}^d$ (for example, to 0)
- 2) Loop for each episode:
- 3) Initialize S , the first state of the episode.
- 4) Sample a state $a \sim \pi(a|S, \theta)$ according to the policy.
- 5) While S is not a terminal state do:
 - a. Observe the results of action a : observe S' , the new state after taking action a at state S , and observe reward R for taking action a at state S .
 - b. Sample an action using the policy: $a' \sim \pi(a'|S', \theta)$ (we sample an action given that we are at state S' and have the parameters θ).
 - c. $\delta = R + \gamma \cdot Q_w(S', a') - Q_w(S, a)$
(where $Q_w(S', a')$ is the Q value estimation according to the value function for state S' and action a' . Same idea for $Q_w(S, a)$.)
 - d. $\theta = \theta + \alpha Q_w(S, a) \nabla_{\theta} \log \pi(a|S, \theta)$
 - e. $w = w + \beta \delta (\nabla_w Q_w(S, a))$ where δ we obtained from step c.
 - f. Update: $a = a', S = S'$

Q: In the algorithm we *sample* actions from the policy. How does this works?

A: The policy is a probability distribution over actions given a state.

The policy defines the likelihood of taking each possible action when in a given state.

The actor may use various mechanisms to sample actions from its policy, such as stochastic policies that introduce randomness, ensuring exploration of different actions.

Resources for section

- LEARNING PYTORCH WITH EXAMPLES, from PyTorch.org
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html
- MODULE documentation, from PyTorch.org
<https://pytorch.org/docs/stable/generated/torch.nn.Module.html>
- DEFINING A NEURAL NETWORK IN PYTORCH, from PyTorch.org
https://pytorch.org/tutorials/recipes/recipes/defining_a_neural_network.html
- David Silver's DeepMind RL course
<https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PLzuuYNsE1EZAXYR4FJ75jcJseBmo4KQ9->
- "Reinforcement Learning: An Introduction", second edition Richard S. Sutton and Andrew G. Barto

Propositional Logic

The purpose of this section is to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we are going to develop is based on *propositions*, or *declarative sentences* which we can argue to be true or false.

Common Symbols:

\neg : The negation of a proposition p is marked $\neg p$

\vee : Denotes a disjunction of two propositions. This symbol is used to create formulas like $p \vee q$, which evaluate to true only if at least one of the propositions - p or q - is true.

\wedge : Denotes a conjunction (=combination) of two propositions. This symbol is used to create formulas like $p \wedge q$, which means p and q must be true for the formula $p \wedge q$ to be true.

\rightarrow : Denotes an implication between two propositions. This symbol is used to create formulas like $p \rightarrow q$, which means that if p is true, then so is q , suggesting that q is a logical consequence of p .

\vdash : We use this symbol to denote that we are trying to use a set of given formulas, which we call premises, to proof another formula, which we call a conclusion. For example: If we have a set of premises: $\phi_1, \phi_2, \dots \phi_n$ and a conclusion: ψ , and if we hope to use the premises to eventually proof the conclusion, we can denote it using a sequent like so: $\phi_1, \phi_2, \dots \phi_n \vdash \psi$. The sequent is consider valid if a proof for it can be found.

\models : We use this symbol to denote that for all valuations in which all $\phi_1, \phi_2, \dots \phi_n$ evaluate to True, ψ evaluates to True as well. We mark it like so: $\phi_1, \phi_2, \dots \phi_n \models \psi$.

\equiv : Used to denote semantically equivalence.

\perp : Used to represent a contradiction.

\top : Used to denote a statement that is a tautology.

Rules for natural deduction:

A rule is written in the following format:

$$\frac{\text{premises}}{\text{conclusion}} (\text{name of the rule})$$

The name of the rule is usually made of a basic symbol and a letter. Above the line are the premises, below the line is the conclusion that can be obtained from the premises using proof rules.

Resources For section:

"LOGIC IN COMPUTER SCIENCE: Modelling and Reasoning about Systems", second edition by Michael Huth and Mark Ryan.

Introduction to NuSMV

NuSMV is a symbolic model checker. It runs from the CMD window.

- 1) After downloading NuSMV, I had a main folder with five sub-folders.

- 2) After opening CMD, the `cd` command can be used to move to the bin folder.
- 3) I used the following command to start NuSmv: `NuSMV - int`
- 4) Writing a NuSmv file: To do that we open a text file using notepad and save it with a `".SMV"` ending in the bin folder.

A string starting with two dashes (`'--'`) and ending with a newline is a comment.

A multiline comment starts with `'/--'` and ends with `'--/'`.

Comparison operators:

`=` `!=` `>` `<` `<=` `>=`

Logic operators:

`&` `|` `xor` `!(not)` `->` `<->`

Conditional expression:

We can write a logical conditional expression "if `cond1` then `expr1` else if `cond2` then `expr2` else if . . . else `exprN`" in NuSMV syntax like so:

`case`

`cond1 : expr1;`

`cond2 : expr2;`

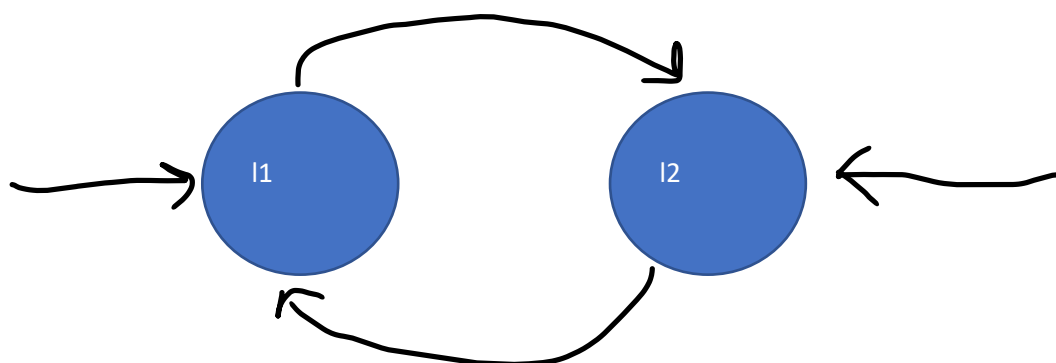
`...`

`TRUE : exprN; -- otherwise`

`esac`

Hands-on example:

In this introduction tutorial we are going to create a simple FSM with two states: `l1` and `l2`. There are arrows between the states described in the image below. State `l1` and state `l2` can both be the initial state.



Compiling the program in interactive mode by typing in CMD:

```
NuSMV > read_model -i trial1.smv
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
```

NuSMV Basic Commands:

1) pick_state -i

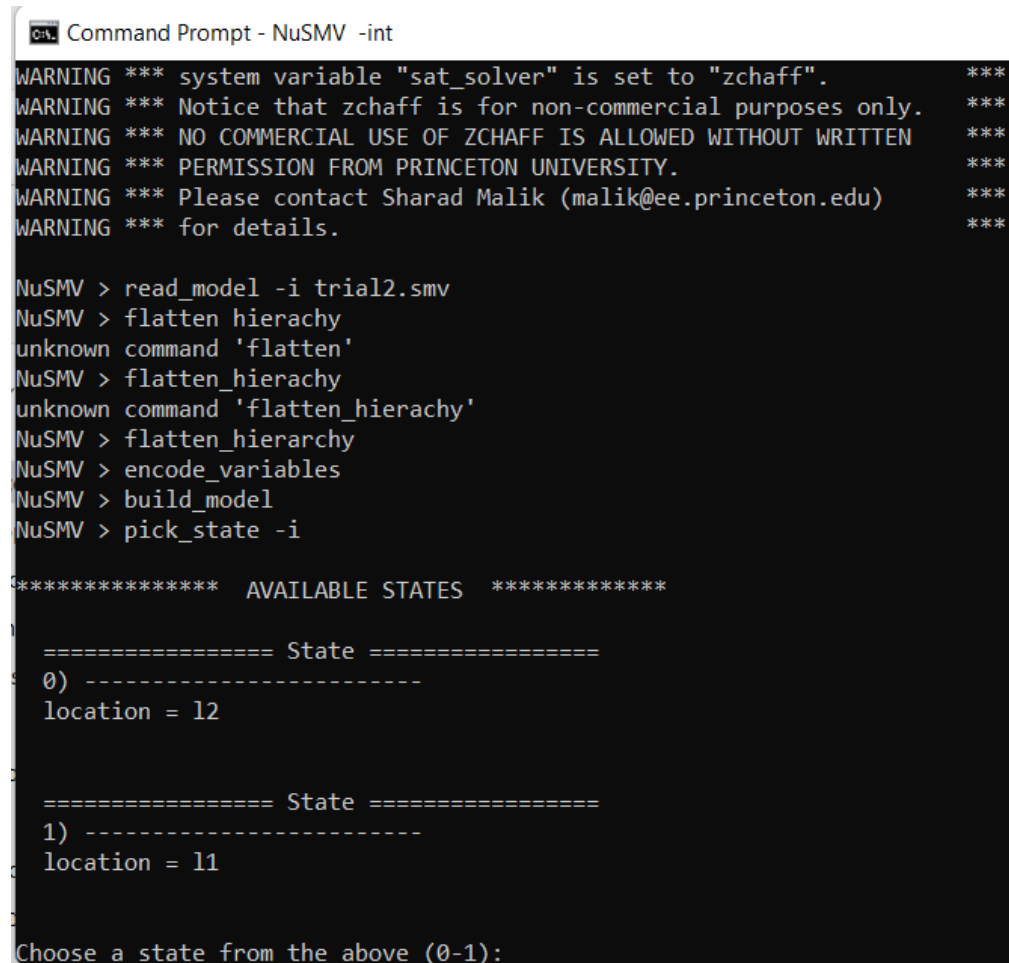
Find all of the initializing states.

- a. Running with our file trial1.NuSMV give us:

We will create a new SMV file named trial2. Our .SMV file would be:

```
MODULE main
VAR
    location: {l1,l2};
ASSIGN
    init(location) := {l1,l2};

    next(location) := case
        location = l1 : l2;
        location = l2 : l1;
    esac;
```



```
Command Prompt - NuSMV -int

WARNING *** system variable "sat_solver" is set to "zchaff". ***
WARNING *** Notice that zchaff is for non-commercial purposes only. ***
WARNING *** NO COMMERCIAL USE OF ZCHAFF IS ALLOWED WITHOUT WRITTEN ***
WARNING *** PERMISSION FROM PRINCETON UNIVERSITY. ***
WARNING *** Please contact Sharad Malik (malik@ee.princeton.edu) ***
WARNING *** for details. ***

NuSMV > read_model -i trial2.smv
NuSMV > flatten_hierarchy
unknown command 'flatten'
NuSMV > flatten_hierarchy
unknown command 'flatten_hierarchy'
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > pick_state -i

***** AVAILABLE STATES *****

===== State =====
0) -----
location = l2

===== State =====
1) -----
location = l1

Choose a state from the above (0-1):
```

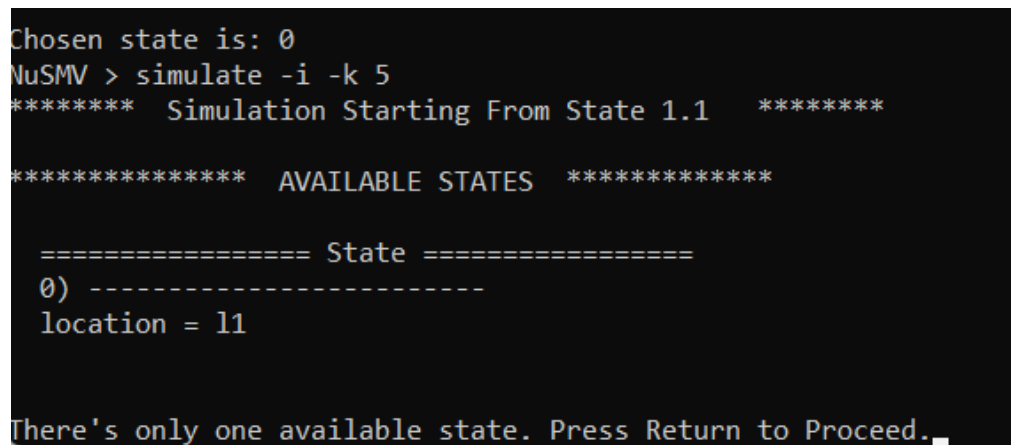
As we can see in the above picture, there are two starting spaces possible. We can pick to start from state l2 by typing 0 and enter. Otherwise, we can type 1.

2) `simulate -i -k n`

Simulating the possible places for the next n steps. n should be a natural number.

Let's simulate the program for 5 steps for trial1.SMV by typing:

```
simulate -i -k 5
```



```
Chosen state is: 0
NuSMV > simulate -i -k 5
***** Simulation Starting From State 1.1 *****

***** AVAILABLE STATES *****

===== State =====
0) -----
location = l1

There's only one available state. Press Return to Proceed.
```

We can continue to walk through our defined model for 5 steps.

Resource for sections:

- “Simple models in NuSMV”, published on youtube:
<https://www.youtube.com/watch?v=GlrOek9sGyQ>
- NuSMV User Manual

Verification of Vanila Q-learning

Previously, students that have done the verification using reinforcement learning have implemented a method to verify vanilla Q-learning algorithm and use it to help algorithm converge.

Our goal is to do so to deep q-learning.

Encoding Q-table as a final state machine

Here I am going to assume that the q-table is encoded.

We do a q-learning step and update table. From the table we can infer which step has the best Q-value for each position we are in. We will create a transition system according to the current policy:

Each position would be connected to the position next to it with the highest Q value.

For example:

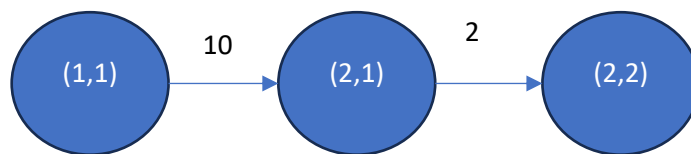
Board:

(1,1) start	(1,2)
(2,1)	(2,2) goal

If the q-table is:

Current position index	Moving left	Moving right	Moving up	Moving down
(1,1)	0	5	0	10
(1,2)	8	0	0	6
(2,1)	0	2	0	0
(2,2)	10	0	8	0

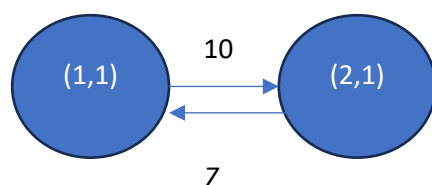
The system to verify:



On the other hand, for:

Current position index	Moving left	Moving right	Moving up	Moving down
(1,1)	0	5	0	10
(1,2)	8	0	0	6
(2,1)	0	2	7	0
(2,2)	10	0	8	0

The system to verify:



And we will get an infinite loop, never reaching the goal.

The Marabou verification tool

Marabou is a neural network verification tool based on satisfiability modulo theories (SMT).

Here is a quick guide to it:

Defining a network

We define a network regularly using Pytorch like so:

```
class QNetwork(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(QNetwork, self).__init__()
        self.layer1 = nn.Linear(n_observations, 16)
        self.layer2 = nn.Linear(16, 16)
        self.layer3 = nn.Linear(16, 4)
        self.layer4 = nn.Linear(4, n_actions)

    # Returns tensor
    def forward(self, x):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
        return self.layer4(x)
```

We create an instance of the network:

```
model = NeuralNet()
```

After completing the desired steps, we save the file as a .nnet file using the torch.save() function. This function takes two arguments: the model to be saved and the file path where you want to save the model.

```
nnetFilePath = "../src/input_parsers/acas_example/NetworkName.nnet"

torch.save(model.state_dict(), nnetFile) #saving just a dictionary containing the model's parameters
and their corresponding values.

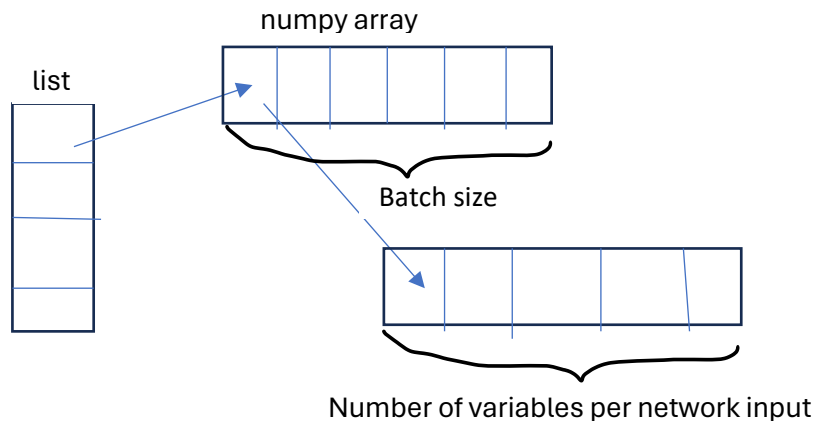
torch.save(model, nnetFile) # saving all the model
```

We can now turn our Pytorch model into a Marabuo nnet network. A Marabuo network devired from a .nnet file is of class MarabouNetworkNNNet [2]. The class MarabouNetworkNNNet inherits from MarabouNetwork class [3] and therefore have its fields.

```
net1 = Marabou.read_nnet(nnetFilePath)
```

Two of the most important fields of the MarabouNetwork object are:

- **inputVars** – which is a list of numpy arrays. To my understanding, the length of each array in the list is the size of the batch, and each array is an array of arrays, but this is not clearly stated at the documentation page.
An array inside an array represents a specific input state.

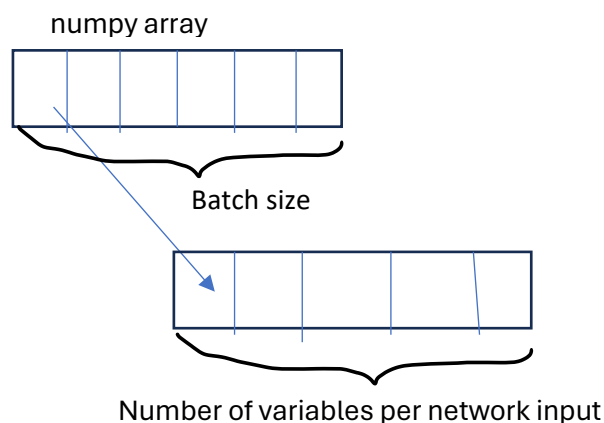


(I think that **inputVars** is a list of arrays of arrays and not simply an array of arrays for support of level-wise input constraints. The length of the list should correspond to the number of neuron layers. In all examples that I found, inputVars was used always with a [0] at the beginning. So, I don't think it is currently supported to be anything else).

Therefore, if we want to refer to the first input state of the network, we will write: `net1.inputVars[0][0]`. If we wish to refer to the first variable in the first state given as an input to the network, we can write: `net1.inputVars[0][0][0]`.

For example: in a frozen lake 3x3 board where the states are encoded as a zeros & ones torch, `net1.inputVars[0][0]` can be `[1,0,0,0,0,0,0,0]`, and `net1.inputVars[0][0][0]` would be simply the `1`.

- **outputVars** – a numpy array. The length of the array is the batch size. To access the first variable of the output, we can write: `net1.outputVars[0][0]`



Making a query

Each query is made out of a conjunction (=and) of inequalities [4].

There are several types of verification queries that Marabou can answer [5] :

- Reachability queries: if inputs is in a given range is the output guaranteed to be in some, typically safe, range.

- Robustness queries: test whether there exist adversarial points around a given input point that change the output of the network.

Properties can be specified as .txt file (via the command line) or using the Python interface, Maraboupy. To run, Maraboupy needs a network encoded in the .nnet format.

Making a query using the command line

The Marabou executable can be called directly from the command line. It takes as arguments the network to verified and the property.

Property grammar:

```
Scalar      ::= <double>
InputVariable ::= x<integer>
OutputVariable ::= y<integer>
HiddenVariable ::= h_<integer>_<integer>           (layer and node index)
Variable    ::= InputVariable | OutputVariable | HiddenVariable
Rel         ::= >= | = | <=
Coefficient  ::= + | - | <double>
Addend      ::= Coefficient Variable               (no separating space)
Comment     ::= //<string>
Line        ::= [Addend] Rel Scalar | Comment
Property    ::= [Line]
```

Note: lines are implicitly connected by a conjunction. There is currently no way to include a disjunction using a single file.

Example: for a network with N+1 inputs and M+1 outputs, we refer to the input variables using x0 to xN and to the output variables as y0 to yM.

```
x4 <= -0.45
y0 >= 3.9911256459
```

Making a query using the python interface

Queries in Marabou are objects of the class **maraboupy.MarabouCore.InputQuery**. However, since we are using a Marabou network extracted from a .nnet format network, we won't have to explicitly create an object of **InputQuery** class. Instead, we can use functions that set queries directly on our MarabouNetwork class object. How does this work?

We pick one of the available functions for queries and give it its inputs:

```
net1.function(inputs)
```

On our Marabuo network `net1`, from the previous section, we can apply a *function* (we will describe some of the popular functions soon), and we give the function the *inputs* it needs.

Examples for possible functions:

- `addInequality(vars, coeffs, scalar)`
Function to add inequality constraint to network.

$$\sum_i (vars)_i \cdot (coeffs)_i \leq scalar$$

- `addEquality(vars, coeffs, scalar)`
Function to add equality constraint to network.

$$\sum_i (vars)_i \cdot (coeffs)_i = scalar$$

- `setLowerBound(x, v)`
Function to set lower bound for variable.
 - `x` is the variable, and can be specified using the `inputVars` or the `outputVars` fields I explained before.
 - `v` is an numerical value. `x` needs to be **equal** or above `v`. [6]
 - Example:
`net1.setLowerBound(net1.outputVars[0][0], .5)`

Disjunctions

To add a disjunction of constraints into the query we want to check, we first need to specify all the different disjunct as a `MarabouCore.Equation` object.

For example, to make sure that a variable `var` is always zero or one, we can write:

```
# eq1: 1 * var = 0
eq1 = MarabouCore.Equation(MarabouCore.Equation.EQ);
eq1.addAddend(1, var);
eq1.setScalar(0);

# eq2: 1 * var = 1
eq2 = MarabouCore.Equation(MarabouCore.Equation.EQ);
eq2.addAddend(1, var);
eq2.setScalar(1);

# ( var = 0 ) ∨ ( var = 1 )
disjunction = [[eq1], [eq2]]
net1.addDisjunctionConstraint(disjunction)
```

(The example is from the Marabou verification page)

Resources for section

[1] "How to Save a Trained Model in PyTorch?" By Saturn Cloud <https://saturncloud.io/blog/how-to-save-a-trained-model-in-pytorch/>

[2] Maraboupy documentation

https://neuralnetworkverification.github.io/Marabou/API/2_MarabouNetworkNNNet.html

[3] Maraboupy documentation

https://neuralnetworkverification.github.io/Marabou/API/1_MarabouNetwork.html

[4] Maraboupy documentation

<https://github.com/NeuralNetworkVerification/Marabou/wiki/Marabou-Input-Formats>

[5] Maraboupy documentation

<https://github.com/NeuralNetworkVerification/Marabou>

[6] "WhichLee - An Attack on Numerical Stability in Machine Learning (CTFSGCTF 2021)"

<https://waituck.sg/2022/01/22/ctfsgctf-2021-whichlee-writeup.html>