Encoding ML Models

Encoding ML Models

We need to embed our ML model into an optimization model

The basic approach is based on two observations:

- A neural network is a collection of connected neurons
- So we just need to encode each neuron using a given optimization method

Let's consider for example a ReLU neuron

This corresponds to the relation:

$$y = \max\left(0, wx + \theta\right)$$

Where $\boldsymbol{w}$ is the (row) vector of weights and $\boldsymbol{b}$ is the bias. It can be encoded by:

- Introducing a variable for each input
- Introducing a variable for the output
- Modeling (e.g. in MILP, or SMT, or CP) the sum and max operators

Encoding ML Models

The type of encoding depends on the considered optimization technology

- In CP, we may have a global constraint per neuron or for a the whole network
- In SMT, we could use ITE (If Then Else) predicates to model the max
- In MILP, we would use linear constraints and both numeric/integer variables

In this case, we will adopt a MILP encoding

The relation $y = max(0, wx + \theta)$ can be translated to:

$$
\begin{aligned}
& y - s = wx + \theta \\
& z = 1 \Rightarrow s \leq 0 \\
& z = 0 \Rightarrow y \leq 0 \\
& y, s \geq 0, x \in \mathbb{R}^n, z \in \{0, 1\}
\end{aligned}
$$

Encoding ML Models

Let's have a better look at the encoding:

$$
\begin{aligned}
& y - s = wx + \theta \\
& z = 1 \Rightarrow s \leq 0 \\
& z = 0 \Rightarrow y \leq 0 \\
& y, s \geq 0, x \in \mathbb{R}^n, z \in \{0, 1\}
\end{aligned}
$$

If $z = 1$, it means that the neuron is active

- In this case s is forced to 0, we have: $y = wx + \theta$
- ...And $wx + \theta$ is non-negative

If $z = 0$, it means that the neuron is inactive

- In this case y (the neuron output) is 0
- We have $s = wx + \theta$ (note that s does not contribute to the neuron output)
- ...And $wx + \theta$ is negative

Loading the Network

We will handle the encoding via the EMLlib

It's a small (and still rough) library for the EML approach. It allows to:

- Load models from ML libraries (currently NNs from keras, DTs from sklearn)
- Convert them into an internal format
- ...And translate such format into an encoding for a target technique

As a first step, let's load both our trained models

In [100]:
```
knn0 = util.load_ml_model('nn0')
knn1 = util.load_ml_model('nn1')
```

...And then convert the in the EMLlib internal format:

In [101]:
```
nn0 = keras_reader.read_keras_sequential(knn0)
nn1 = keras_reader.read_keras_sequential(knn1)
```

Loading the Network

Printing the loaded network shows "bounds" for each neuron

This is easier to parse for the Linear Regression model:

```
In [102]: nn0

Out[102]: [input] (0, 0):[-inf, inf] (0, 1):[-inf, inf] (0, 2):[-inf, inf] (0, 3):[-inf, inf]
          [dense,linear] (1, 0):[-inf, inf]/[-inf, inf] (1, 1):[-inf, inf]/[-inf, inf] (1, 2):[-inf, in
          f]/[-inf, inf]
```

The bounds represent the domain of output of each neuron

- Currently, most output have an infinite range
- ...Since we have not specified a finite range for the network input

This is a problem since our MILP encoding for each neuron...

...Requires finite bounds to linearize the indicator constraints

- Actually, the tighter the bounds, the better the MILP encoding will work

Loading the Network

We can use 0 and 1 as bounds for all our inputs

...Since the population is normalized and β is typically a low value

```
In [103]: nn0.layer(0).update_lb(np.zeros(4))
          nn0.layer(0).update_ub(np.ones(4));
```

Internal bounds can be inferred with one round of constraint propagation

- This needs to be done for the weighted sum in each neuron
- ...And for the ReLU, when actually present

The process is implemented in the `ibr_bounds` function:

```
In [104]: from eml.net.process import ibr_bounds
          ibr_bounds(nn0)
          nn0
```

```
Out[104]: [input] (0, 0):[0.000, 1.000] (0, 1):[0.000, 1.000] (0, 2):[0.000, 1.000] (0, 3):[0.000, 1.00
          0]
          [dense,linear] (1, 0):[-0.681, 0.849]/[-0.681, 0.849] (1, 1):[-0.319, 0.997]/[-0.319, 0.997]
          (1, 2):[0.030, 1.305]/[0.030, 1.305]
```

Network and I/O Variables

We will consider a planning problem over eoh weeks

So, we will create eoh distinct encodings of the same network

- Each will connect variables representing S, I, R, β for week t
- ...With variables representing S, I, R for week $t + 1$, i.e.:

$$
\begin{aligned}
&(S_{t+1}, I_{t+1}, R_{t+1}) = NN(S_t, I_t, R_t, \beta_t) && \forall t = 0..eoh - 1 \\
&\beta_t \in [0, 1] && \forall t = 0..eoh - 1 \\
&S_t, I_t, R_t \in [0, 1] && \forall t = 0..eoh
\end{aligned}
$$

Where $NN(\ldots)$ represents the network encoding

- We need extra S, I, R variables to represent the final state
- The S_0, I_0, R_0 will be fixed to the values from the initial state

Our objective will be to maximize S_{eoh}

Network ad I/O Variables

The code for the planning problem is in `solve_sir_planning`

We use the CBC solver via Google Or-tools:

```
slv = pywraplp.Solver.CreateSolver('CBC')
```

We start by building the network I/O variables:

```
for t in range(nweeks+1):
    X['S', t] = slv.NumVar(0, 1, f'S_{t}')
    X['I', t] = slv.NumVar(0, 1, f'I_{t}')
    X['R', t] = slv.NumVar(0, 1, f'R_{t}')
    if t < nweeks: X['b', t] = slv.NumVar(0, 1, f'b_{t}')
```

- The network will be embedded as an encoding
- ...Which cannot be defined unless we have the variables first

Network Encodings

The EMLlib handles multiple target solver via "backend" objects

Therefore we need to build a backend for Or-tools:

```
bkd = ortools_backend.OrtoolsBackend()
```

- The backend defines the primitives to build the NN constraints

The encoding themselves are built using the `encode` function:

```
for t in range(1, nweeks+1):
    vin = [X['S',t-1], X['I',t-1], X['R',t-1], X['b',t-1]]
    vout = [X['S',t], X['I',t], X['R',t]]
    encode(bkd, nn, slv, vin, vout, f'nn_{t}')
```

- Neurons are processed one by one
- Intermediate variables are built as needed

Non-Pharmaceutical Intervention

Now we need to setup the rest of the optimization model

...Since we delayed this even too much to focus on the NN encoding

- At each week we can choose to apply a number of NPIs
- ...Which stands for "Non-Pharmaceutical Interventions"

Each NPI i has a (socio-economical) cost c_i

- ...And can reduce the current β value by a factor r_i
- β has a "base value", which depends on the disease itself

So, if we apply NPIs 1, 3, and 4:

- We pay a cost equal to $c_1 + c_3 + c_4$
- And we have $\beta = r_1 r_3 r_4 \beta_{base}$

We assume the total cost cannot exceed a maximum budget

Non-Pharmaceutical Intervention

This part of the problem can be formalized as follows:

We introduce a binary variable x_{it} for each NPI and week (except the last)

$$x_{it} \in \{0, 1\} \qquad \forall i = 1 \ldots n_{npi}, \forall t = 0 \ldots eoh - 1$$

- $x_{it} = 1$ iff we apply NPI i at week t

The budget constraint can then be stated as:

$$\sum_{t=0}^{eoh-1} \sum_{i=1}^{n_{npi}} c_i x_{it} \leq C$$

- Where C is the budget value

Non-Pharmaceutical Intervention

The effect on β is non-linear and trickier to handle

We linearize it by introducing multiple variables for β at each week

- β_{0t} represents the "base" β value
- β_{it} represents β as affected by the i-th NPI
- Therefore $\beta_{n_{npi},t}$ is the same as the variable connected to the NN for week t

For each intermediate variable we have:

$$\beta_{it} \geq r_i \beta_{i-1,t} - 1 + x_{it} \qquad \forall i = 1..n_{npi}, \forall t = 0..eoh - 1$$
$$\beta_{it} \geq \beta_{i-1,t} - x_{it} \qquad \forall i = 1..n_{npi}, \forall t = 0..eoh - 1$$

- If $x_{it} = 1$, the first constraint is active and the second is trivialized
- If $x_{it} = 0$, the opposite is true

Non-Pharmaceutical Intervention

An analogous set of constraints handles the upper bounds

$$
\begin{aligned}
\beta_{it} &\leq r_i \beta_{i-1,t} + 1 - x_{it} && \forall i = 1..n_{npi}, \forall t = 0..eoh - 1 \\
\beta_{it} &\leq \beta_{i-1,t} + x_{it} && \forall i = 1..n_{npi}, \forall t = 0..eoh - 1
\end{aligned}
$$

Together with the previous set:

$$
\begin{aligned}
\beta_{it} &\geq r_i \beta_{i-1,t} - 1 + x_{it} && \forall i = 1..n_{npi}, \forall t = 0..eoh - 1 \\
\beta_{it} &\geq \beta_{i-1,t} - x_{it} && \forall i = 1..n_{npi}, \forall t = 0..eoh - 1
\end{aligned}
$$

...We obtain the desired behavior, e.g.:

- If $x_{1t} = x_{3t} = x_{4t} = 1$
- ...Then $\beta_{n_{npi}t} = r_1 r_3 r_4 \beta_{0,t}$

The details of the code can be found in the `solve_sir_planning`

Solving the NPI Planning Problem

We will consider the following set of NPIs

In [105]:
```
npis = [
    util.NPI('masks-indoor', effect=0.75, cost=1),
    util.NPI('masks-outdoor', effect=0.9, cost=1),
    util.NPI('dad', effect=0.7, cost=3),
    util.NPI('bar-rest', effect=0.6, cost=3),
    util.NPI('transport', effect=0.6, cost=4)
]
```

We will consider a horizon of 3 weeks and the following parameters:

In [106]:
```
S0, I0, R0 = 0.99, 0.01, 0.00
nweeks = 3
tlim = 30
beta_base = 0.35
budget = 20
gamma = 1/14
```

- β has a very high value (to better see the impact of our decisions)

Solution and Evalution

Let's start by using the (much more accurate) NN model

In [107]:
```
%%time
sol, closed = util.solve_sir_planning(knn1, npis, S0, I0, R0, beta_base=beta_base, budget=budget
                                      nweeks=nweeks, tlim=tlim)
print(f'Problem closed: {closed}')
sol_df = util.sol_to_dataframe(sol, npis, nweeks)
sol_df
```

```
Problem closed: True
CPU times: user 978 ms, sys: 20 ms, total: 998 ms
Wall time: 998 ms
```

Out[107]:

	S	I	R	b	masks-indoor	masks-outdoor	dad	bar-rest	transport
0	0.990000	0.010000	0.000000	0.0945	1.0	0.0	0.0	1.0	1.0
1	0.985713	0.016681	0.002336	0.0945	1.0	0.0	0.0	1.0	1.0
2	0.973421	0.025539	0.008058	0.1575	1.0	0.0	0.0	1.0	0.0
3	0.909765	0.068966	0.028244	NaN	NaN	NaN	NaN	NaN	NaN

- The result seem pretty good
- ...But are we sure about that?

Solution and Evalution

Our optimization model relies on predictions

We need to test their quality on the simulator:

```
In [108]: beta_sched = sol_df.iloc[:-1]['b']
          util.simulate_SIR_NPI(S0, I0, R0, beta_sched, gamma, steps_per_day=1)
```

Out[108]:

	S	I	R
0	0.990000	0.010000	0.000000
1	0.982948	0.011649	0.005404
2	0.974816	0.013501	0.011683
3	0.955472	0.023756	0.020773

Unless we've been unlucky during training (it's stochastic!)

- The final value for S should be close to 0.95
- ...And possibly quite different from our model predictions!

Solution and Evalution

It's even more clear if we use the Linear Regression model

In [109]:
```
%%time
sol2, closed2 = util.solve_sir_planning(knn0, npis, S0, I0, R0, beta_base=beta_base, budget=budg
                                        nweeks=nweeks, tlim=tlim)
print(f'Problem closed: {closed}')
sol_df2 = util.sol_to_dataframe(sol2, npis, nweeks)
sol_df2
```

```
Problem closed: True
CPU times: user 770 ms, sys: 80 ms, total: 850 ms
Wall time: 850 ms
```

Out[109]:

	S	I	R	b	masks-indoor	masks-outdoor	dad	bar-rest	transport
0	0.990000	0.010000	0.000000	0.1575	1.0	0.0	0.0	1.0	0.0
1	0.765613	0.181725	0.051822	0.0945	1.0	0.0	0.0	1.0	1.0
2	0.606383	0.235049	0.157822	0.0945	1.0	0.0	0.0	1.0	1.0
3	0.487314	0.235834	0.276197	NaN	NaN	NaN	NaN	NaN	NaN

- Now the solution process is very fast
- ...And it looks like a disaster

Solution and Evalution

However, if we evaluate the solutions via the simulator...

In [110]:
```
beta_sched2 = sol_df2.iloc[:-1]['b']
util.simulate_SIR_NPI(S0, I0, R0, beta_sched2, gamma, steps_per_day=1)
```

Out[110]:

	S	I	R
0	0.990000	0.010000	0.000000
1	0.975265	0.017934	0.006801
2	0.962909	0.020653	0.016438
3	0.948935	0.023577	0.027488

...They are not bad at all!

Our ML models are making mistakes

- For many reasons: bias, compound error, "weak spots"
- ...But as long as they guide the solver in the right place, we get a good solution

This is good news, but leave some open issues (see later)

Solution and Evaluation

The main issue is: how much can we trust our models?

In our case, it turns out the answer is "a lot"

- Here's what we get by solving the problem via brute force:

```
In [111]: %%time
          best_S, best_sched = util.solve_sir_brute_force(npis, S0, I0, R0, beta_base, gamma, nweeks, budg
          best_S

          CPU times: user 39.9 s, sys: 5.71 ms, total: 39.9 s
          Wall time: 39.9 s

Out[111]: 0.9554715100410379
```

- The NN solution in particular is almost perfect
- ...And we obtain it in a much less time
- As the problem size grows, the gap in computation time becomes larger

Some Considerations

This kind of hybrid approach can be complex to build

- But sometimes it's (almost) the only choice!
- It generally worked in our case

EML-like approaches can be used to generate adversarial examples

- It is at the basis of some tools for NN verification

There are several open issues

- The optimizer often ends up finding weaknesses in the ML model
  - Can we bound the error, or define confidence intervals?
  - Can we use re-train the model to fix mistakes?
- The approach scalability is limited
  - Large (even moderately large) ML models are currently out of reach
  - How to improve that?