# 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**

$$y = \max\left(0, w^T x + \theta\right)$$

Where $w$ is the (row) vector of weights and $b$ 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

**In this case, we will adopt a MILP encoding for the relation**
$y = max(0, wx + \theta)$

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

- $s$ is an auxiliary slack variable and $z$ is an auxiliary binary variable
- The implications are called indicator constraints
- They are handled natively by some MILP solvers, or they can be linearized

# 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

# 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 $z = 0$, 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 [35]:
```
knn0 = util.load_ml_model('nn0')
knn1 = util.load_ml_model('nn1')
```

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

In [36]:
```
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 [37]: nn0

Out[37]: [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

## 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 $\beta$ is typically a low value

```
In [38]: 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 [43]: from eml.net.process import ibr_bounds
         ibr_bounds(nn0)
         nn0
```

## Network and I/O Variables

**We will consider a planning problem over $eoh$ weeks**

So, we will create variables to represent $S$, $I$, $R$, and $\beta$ for those weeks:

$$\beta_t \in [0, 1] \qquad \forall t = 0..eoh - 1$$
$$S_t, I_t, R_t \in [0, 1] \qquad \forall t = 0..eoh$$

- For week $t$ we do not need to represent the value of $\beta$
- ...But we need the $S$, $I$, $R$ variables to represent the final state
- $S_0$, $I_0$, $R_0$ will be fixed to the values from the initial state

**Our objective will be to maximize $S_{eoh}$**

# Network and I/O Variables

**We will encode an instance of our network for each week**

Each will connect consecutive values of $S, I, R$

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

- $\mathrm{NN}(\ldots)$ represents the network encoding
- ...I.e. the equations we discussed early on in the lecture
- Each network takes as input the value of $S$, $I$, $R$, and $\beta$ for week $t$
- ...And links them to the value of $S$, $I$, $R$ for week $t + 1$

# 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 library we use handles multiple solvers 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 (we remind) stands for "Non-Pharmaceutical Interventions"

**We will assume each NPI $i$ has a (socio-economical) cost $c_i$**

- ...And can reduce the current $\beta$ value by a factor $r_i$
- $\beta$ 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 $\beta = r_1 r_3 r_4 \beta_{base}$

**Using multiple NPIs has diminishing returns**

# 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 $t$

**We assume the total cost should not exceed a given budget**

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

- Where $C$ is the budget value

# Non-Pharmaceutical Intervention

**The effect on $\beta$ is non-linear and trickier to handle**

We linearize it by introducing multiple variables for $\beta$ at each week

- $\beta_{0t}$ represents the "base" $\beta$ value
- $\beta_{it}$ represents $\beta$ 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 [46]:
```
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 [48]:
```
S0, I0, R0 = 0.99, 0.01, 0.00
nweeks = 3
tlim = 30
beta_base = 0.35
budget = 20
gamma = 1/14
```

# Solution and Evalution

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

In [55]:
```
%%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 1.57 s, sys: 60.8 ms, total: 1.63 s
Wall time: 1.63 s
```

Out[55]:

| | S | I | R | b | masks-indoor | masks-outdoor | dad | bar-rest | transport |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.990000 | 0.010000 | 0.000000 | 0.14175 | 1.0 | 1.0 | 0.0 | 1.0 | 0.0 |
| 1 | 0.944169 | 0.014658 | 0.010808 | 0.09450 | 1.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 2 | 0.929573 | 0.010191 | 0.023377 | 0.11025 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 |
| 3 | 0.912024 | 0.017791 | 0.036351 | NaN | NaN | NaN | NaN | NaN | NaN |

- The result seem reasonable
- ...But how can we know for sure?

# Solution and Evalution

## Our optimization model relies on predictions

We need to test their quality on the simulator:

In [57]:
```
beta_sched = sol_df.iloc[:-1]['b']
util.simulate_SIR_NPI(S0, I0, R0, beta_sched, gamma, steps_per_day=1)
```

Out[57]:

| | S | I | R |
|---|---|---|---|
| **0** | 0.990000 | 0.010000 | 0.000000 |
| **1** | 0.977482 | 0.016106 | 0.006412 |
| **2** | 0.966344 | 0.018582 | 0.015075 |
| **3** | 0.950792 | 0.023622 | 0.025586 |

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 [58]:
```
%%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 487 ms, sys: 33.3 ms, total: 521 ms
Wall time: 521 ms
```

Out[58]:

| | 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** | 0.765027 | 0.183567 | 0.053728 | 0.0945 | 1.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **2** | 0.608145 | 0.235826 | 0.160640 | 0.0945 | 1.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| **3** | 0.492628 | 0.235837 | 0.278775 | 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 [59]:
```
beta_sched2 = sol_df2.iloc[:-1]['b']
util.simulate_SIR_NPI(S0, I0, R0, beta_sched2, gamma, steps_per_day=1)
```

Out[59]:

| | 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 leaves 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 [62]: %%time
         best_S, best_sched = util.solve_sir_brute_force(npis, S0, I0, R0, beta_base, gamma, nweeks, budg
         best_S

         CPU times: user 25.9 s, sys: 0 ns, total: 25.9 s
         Wall time: 25.9 s

Out[62]: 0.9554715100410379
```

- The NN solution in particular is actually pretty good
- ...And we obtain it in 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
- The approach scalability is limited