1 Single Perceptron

Using the perceptron model illustrated in Fig 1. Choose a weight vector which will make the perceptron capable to replicate AND, OR, NAND, and NOR respectively. The inputs x_1, x_2 and the output y are binary.

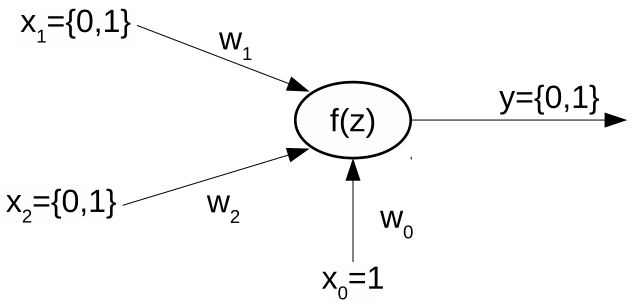

Figure 1: Single perceptron with two inputs and bias

$$z = \sum_{i=0}^{2} w_i x_i \qquad f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1

AND

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	1

OR

x_1	x_2	y
0	0	1
1	0	1
0	1	1
1	1	0

NAND

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	0

NOR

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

XOR

1.1 Solution

- For the AND operation we need the neuron to be activated only when it receives two inputs at the same time($x_1 = x_2 = 1$). Thus we can bias negatively the node with such a bias weight w_0 that allows $z \geq 0$ only if both inputs are received. Such a possible set of weights would be $w_0 = -2, w_1 = 1, w_2 = 1$. You can get the analytical solution by writing down the equations for each of the four possible combinations of inputs-outputs (see the AND table)

- For the OR operation we need the neuron to be activated when at least one of the inputs is nonzero. Thus, we can set the bias weight to a negative value. Also, we want $z \geq 0$ with at least one input. Thus a possible solution would be $w_0 = -1, w_1 = 1, w_2 = 1$. What we change here compared to the AND solution is that we lower the threshold for which the neuron is activated. This allows its activation with fewer inputs.

- For the NAND case, the neuron has to be deactivated only when it receives two inputs. Thus, we use a positive weight for the bias and negative weights for the other inputs. A possible solution would be: $w_0 = 1, w_1 = -1, w_2 = -1$

- At the NOR case, the neuron has to be deactivated when it receives at least one input. This can be achieved by setting the weights such as: $w_0 = 0.5, w_1 = -1, w_2 = -1$
- At the XOR case is not feasible.

2 Single perceptron without bias

Consider a single perceptron without bias. Given a set of $n = 4$ samples $X = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$ in 3-dimensional space, where $\mathbf{x}_1 = [1, 0, 0]^T$, $\mathbf{x}_2 = [0, 1, 0]^T$, $\mathbf{x}_3 = [0, 0, 1]^T$, $\mathbf{x}_4 = [1, 1, 1]^T$, and the corresponding desired outputs y_1, y_2, y_3, y_4. Find any parameter vector $\mathbf{w}$ such as:

1. $y_1 = y_2 = y_3 = y_4 = 1$
2. $y_1 = y_2 = 1$ and $y_3 = y_4 = 0$
3. Find a set of values y_1, y_2, y_3, y_4 which the perceptron cannot realize for the given inputs $\mathbf{x}_n$
   Hint: Consider the problem as classification where inputs are $\mathbf{x_n}$ and labels are y_n

Use the same activation function $f(\cdot)$ as in Exercise 1.

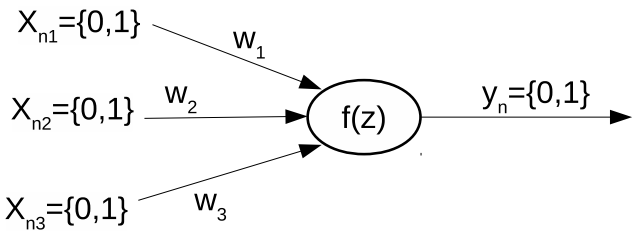

Figure 2: Single perceptron taking three-dimensional inputs, without bias

2.1 Solutions

1. In the first case we want the perceptron to fire for any set the inputs. This is achieved by setting a set of zero or positive weights for all the three inputs, for example $w_1 = w_2 = w_3 = 1$.
2. In the second case, the perceptron should be activated only if it receives values from either the first or second input. This can be achieved by putting a small positive weight at those two and a large negative weight at the third. A possible solution would be $\mathbf{w} = [0.25, 0.25, -1]$
3. If you consider our problem in terms of classification, the perceptron creates a plane in 3d space which has to seperate the two classes ($y = \{0, 1\}$). Given that our perceptron does not have a bias term, the boundary that learns has to pass through the axes origin $(0, 0, 0)$. Thus, even though the dataset is seperable the specific model we consider here, cannot find a solution since the boundary has to pass through the axes origin. The two sets of unrealized values are $y = [0, 0, 0, 1]$ and $y = [1, 1, 1, 0]$

3 Computational Cost of Neural Networks

1. **Open Question** [The solution to this question is not needed for the rest of the exercise.]
   You are employed by a big software company and are asked to create a neural network that is able to play the game of Go (see Fig. **??**). The go board consists of a board with a grid of 19x19, and is played by two opponents (black and white). Each turn a player can either place a stone or pass. The goal of the game is to *encircle* the opponent.
   How could you design an artificial neural network that can be used to play the game of GO?

2. You ended up choosing a neural network with input and output sizes of 361 each, additionally, it has 12 hidden layers each of size 722.
   How many parameters does your model have? How much memory is needed to store all of them? (Assume all parameters to be double values, i.e., 8 bytes on a 64-bit system.)

3. You want to take expert games to train and test your network. Let us assume that a game of GO has on average 200 moves. You use each move of the winner as one data point. You want a train-test ratio of 70:30, and you want to have at least $500N$ samples to N parameters for the training. How many games are required for the training and testing?

4. How many operations will be needed for the inference[1] of the network? How fast could this be evaluated on a 4GHz CPU (we simplify that one mathematical operation can be executed in 5 clock cycles, including loading and storing).

5. The large computational cost and time stem from training these networks. The neural network-based Go-engine AlphaGo[2] used supposedly 50 GPUs for 3 weeks. Let us assume they used GPUs with an average power consumption of 200 W. How much energy did the training consume? Compare this to the average per capita electricity consumption of a Switzerland (6721 kWh / year).

6. In 2020, the natural language model GPT-3[3] used around 1000 GPUs for 34 days to train the network. How does this compare to the AlphaGo model? How do you explain the difference?

3.1 Solution

1. A possible way is to use the input of an array of the shape 361(=19*19), which has the values of either:

   - 1: stone of the winner)
   - 0: free position
   - -1: stone of the opponent

   The output is again of length 361, and it predicts the probability of placing a stone at a specific position (note that the maximum of the free position has to be taken).
   We can then train the neural network by observing professional games and using the *move* of the winner of a match as a data point. The input vector $X \in \mathbb{R}^{361}$ is the current state of the

[1] Inference is the process of running data points into a machine learning model to calculate an output such as a numerical score.

[2] `https://en.wikipedia.org/wiki/AlphaGo`

[3] `https://en.wikipedia.org/wiki/GPT-3`

game, and the output vector $Y \in \mathbb{R}^{361}$ is a zero-vector with a value of one at the position the stone was placed.

2. All the weights can be calculated as: $N_w = 361 * 722 + 722 * 722 * 11 + 361 * 722$
   All the bias are given as $N_b = 722 * 12 + 361$
   Hence the total parameters: $N_e = 6.26e6$ and hence storage required of $47.79MB$.

3. The data points per game is: $100 = 200/2$, the $t_{ratio} = 0.7$. Hence, the total number of games is calculated as $N_g = N_e/t_{ratio} * 500/100 \approx 4.4e7$

4. Additional to the number of parameters being applied once ($w * x$, once for each weight), we also have the adding of each $w * x$ at each node (once for each weight), the adding of the bias (once for each bias), and the rectifier operation (deemed as one operation for each node):
   $N_c = 2 * N_w + 2 * N_b = 12.52e6$.

   The minimal computational evaluation is then expected to be at
   $N_c = 12.521e6/(4e9Hz) = 15.65ms$

5. The power consumption is $P_t = 50 * 21 * 24 * 200Wh = 5050kWh$. The (one time) training of the network is around 75% of the **annual** per capita electricity consumption of Switzerland.

6. The large language model used around $P_t = 1000 * 34 * 24 * 200Wh = 163.3MWh$, this is equivalent to around the annual electricity consumption of 24.3 Swiss citizens.
   It is an increase of x32. The increased complexity can be appointed to the fact that the environment is much more complex. While the Go-board has a clearly defined 19x19 size, the real world and speech / natural language that we use have much larger complexity. As RNN, additionally the network has to keep long and short-term memory. These constraints make the training more difficult and expensive. Moreover, the dataset was much more extensive with more than 500 billion books and web pages; the resulting network ended up having 175 billion parameters.

4 Case Study Robotics: Comparison of Algorithms

You want to learn a model to predict the self-collision of a robot. The input is the joint positions, and the model predicts collisions (value=1) or predicts when the robot is in a collision-free configuration (value=-1). The training data points are obtained from a simulator. Assume a power consumption of 125 W for the GPU and 50 W for the CPU
You ask one of your engineers to train different models. The engineer comes back with the results below:

Learning Method	SVM	NN
Model Size [# of doubles]	2'099'442	15'333
Total training time [s]	2106 (GPU)	11250 (GPU)
True Positive Rate (TPR) [ ]	0.995	0.940
True Negative Rate (TNR) [ ]	0.960	0.990
Inference time [ms]	1.38 (GPU)	0.11 (CPU)

Table 1: Comparison of performance and energy consumption of two classifiers on collision vs. no-collision dataset.

1. Compare the energy usage for training between the two algorithms.
2. Compare the energy usage for inference of the algorithm.
3. Which algorithm would you choose? Justify your choice.

4.1 Solution

1. For **training** we have: SVM used $2106s * 125W/(3600s/h) = 73.1kWh$ for training, the NN $11250s * 125W/(3600s/h) = 390.6kWh$. NN uses 5.3x the amount of energy that the SVM uses.

2. For **inference** we have:
   NN used $0.11s * 50W = 5.50J = 1.53mWh$ for training, the SVM $1.38s * 125W = 172.5J = 47.98mWh$. SVM uses 31.36x the amount of energy that NN uses.

3. The NN requires more time to train, however since this has to be done only few times, as the robot geometry does not change, the time of around ¡ 3 hours is not a problem.
   The SVM has a larger memory usage, however, the size in the order of 10 MB can be stored by modern hardware.
   The inference time is much lower by the NN. This is important in a real-time control system, as depending on the algorithm the collision check needs to be executed several times in a control step. So this is a point for the NN.
   While the SVM has a higher TPR, the NN has a higher TNR. Let us look at what the false classifications imply. A False Positive, is a detection of a collision, where there is none. This might lead to undesired behavior but is not critical. On the other hand, a False Negative implies no detection of a collision, when the robot is in a collision state. This can lead to crashes or accidents. Hence it is more important to have a higher TNR. Another point for the NN.
   As a result. we chose the NN network for the application on the robot.

5 Two-layer Feed-Forward Neural Network [Optional]

Find a set of weights for the two-layer feed-forward neural network (Fig. 3) for which it can learn the XOR function. Consider binary inputs ($x_i = \{0, 1\}$) and output ($y = \{0, 1\}$). Using the same activation function $f(\cdot)$ as in Exercise 1, the nodes are activated by the following equations:

$$h_n = f\left(\sum_{i=1}^{2} w_{in} x_i\right) \qquad y = f\left(\sum_{i=1}^{2} w_{iy} h_i\right) \qquad f(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

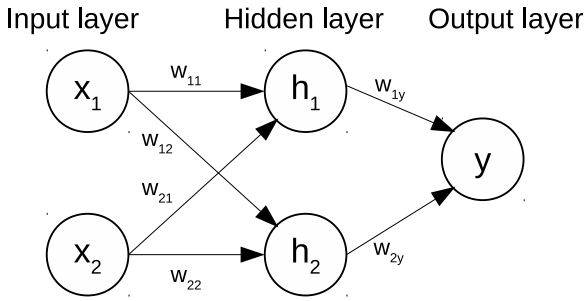

Figure 3: Feed-Forward ANN with a hidden layer

5.1 Solutions

- For XOR function you want to classify the options where only one input node is activated (x_1 = 1, x_2 = 0 and x_1 = 0, x_2 = 1) with y = 1, and the other options (x_1 = 0, x_2 = 0 and x_1 = 1, x_2 = 1) with y = 0 (check XOR table in exercise 1). This requires you to have a neural network with two-layers (one hidden layer) in order to represent a non-linear classifier.

  To represent a XOR function you can use the other functions studied in exercise 1. The OR and the NAND function can be used as the connections between the input layer and the hidden layer. It will fulfill the requirements for the y = 1 of the XOR function. However, to comply with the requirements of y = 0 of the XOR function the connection from the hidden layer to the output layer is a AND function. This allows for the common outputs of hidden layers to be 1 and the uncommon outputs of the hidden layers to be classified as 0.

  The weights for the two-layer feed-forward network are the weights computed in exercise 1 for the function OR, NAND, and AND:

- $w_{11} = w_{21} = 1$; w_{01} = -1
- $w_{12} = w_{22}$ = -1; $w_{01} = 1$
- $w_{1y} = w_{2y}$ = -1; $w_{0y} = 2$

6 Recurrent Neural Network (RNN) [Optional]

Find a set of weights for the RNN illustrated below such that it exhibits oscillating behavior (i.e. the values of the nodes alternate continuously between 0 and 1). Consider that the two nodes are activated asynchronously as:

$$x_2^{t+1} = f(\sum w_{12}x_1^t + w_{22}x_2^t) \qquad x_1^{t=0} = 1 \qquad f(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$$

$$x_1^{t+1} = f(\sum w_{21}x_2^t + w_{11}x_1^t) \qquad x_2^{t=0} = 0$$

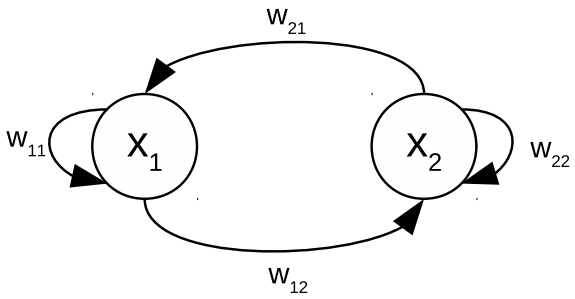

Figure 4: Recurrent Neural Network

6.1 Solutions

- To find the weights so that the values of the nodes alternate continuously over time the values should be:
- $x_1^1 = 0$, $x_2^1 = 1$
- $x_1^2 = 1$, $x_2^2 = 0$

  From the equations of the transfer function of each node we get:
- $w_{11} = w_{22} = -1$
- $w_{12} = w_{21} = 1$