Homework 1

Aneesh Soni
Prof. Alex Tuzhilin

Exercise 1

In [1]:
```
import numpy as np

def relu(x):
    return max(0, x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def calculate_network(x1, x2, x3):
    x4 = relu(3 * x1 + 0 * x2 + 1 * x3)
    x5 = relu(-2 * x1 + 3 * x2 + 3 * x3)
    x6 = relu(1 * x1 + 1 * x2 + 2 * x3)
    x7 = sigmoid(-2 * x4 + 2 * x5 + 3 * x6)

    return x4, x5, x6, x7

# (X1, X2, X3)
input_values = [
    (0, 0, 1),
    (0, 1, 0),
    (1, 0, 0),
    (1, 1, 1),
    (1, -1, 1),
    (1, 1, -1),
    (-1, -1, 1),
    (-1, -1, -1)
]

for inputs in input_values:
    x4, x5, x6, x7 = calculate_network(*inputs)
    print(f"Input: {inputs}")
    print(f"Output: X4 = {x4}, X5 = {x5}, X6 = {x6}, X7 = {x7:.3f}")
    print()
```

```
Input: (0, 0, 1)
Output: X4 = 1, X5 = 3, X6 = 2, X7 = 1.000

Input: (0, 1, 0)
Output: X4 = 0, X5 = 3, X6 = 1, X7 = 1.000

Input: (1, 0, 0)
Output: X4 = 3, X5 = 0, X6 = 1, X7 = 0.047

Input: (1, 1, 1)
Output: X4 = 4, X5 = 4, X6 = 4, X7 = 1.000

Input: (1, -1, 1)
Output: X4 = 4, X5 = 0, X6 = 2, X7 = 0.119

Input: (1, 1, -1)
Output: X4 = 2, X5 = 0, X6 = 0, X7 = 0.018

Input: (-1, -1, 1)
Output: X4 = 0, X5 = 2, X6 = 0, X7 = 0.982

Input: (-1, -1, -1)
Output: X4 = 0, X5 = 0, X6 = 0, X7 = 0.500
```

Exercise 2

Part a)

Under supervised learning, the algorithm is given a set of input-output pairs and learns to map inputs to outputs by adjusting its parameters.

1. Initialize Weights and Biases

A neural network will use random values for the weights and biases with the expectation that the optimization algorithm will adjust the weights and biases to minimize the error between the predicted output and the true output.

2. Forward Propagation

With the values of the weights and biases initialized, the neural network calculates the output of the network for a given input. This is done by multiplying the input by the weight adding any bias to it (if applicable) and feeding it through an activation function. For instance, for exercise 2 part (b) this would look like: Hidden Layer 1: relu(w1 * x1 + w3 * x2). Similarly, it would calculate the second hidden layer and feed those as inputs into the Output Layer: w5 * H1 + w6 * H2. Since the activation function of the output layer is the identity function no further calculations are computed.

3. Loss Function

Since this is a supervised learning model we measure the difference between the predicted output (aka our calculated value) and the expected output (aka the actual value). For this exercise the loss function is the mean squared error.

4. Backward Propagation

Now that we have a value for the loss function we need to update the weights and biases in the direction of the negative gradient of the loss function. Without getting too technical we can achieve this by calculating the partial derivative of the loss function in the negative direction. These gradients ultimately tell us the slope of the loss function at a particular point (indicating how much a minor change in the weights and biases will affect the loss function).

5. Update Weights and Biases

Using the gradients calculated in step 4 we can use gradient descent to adjust the weights towards the direction that minimizes the loss function. This also requires what's known as a learning rate which decides how "aggressively" or "slowly" we want to be in our adjustments.

6. Rinse and Repeat

Steps 1-5 are repeated for a number of iterations (epochs) or until the loss function is minimized to an acceptable level. Once all iterations are done the neural network will have learned to the best of its ability the "ideal" weights and biases to make predictions on new data.

Part b)

Let's call the top hidden layer H1, the bottom hidden layer H2 and the calculated output layer O1

In [28]:
```
def relu(x):
    return max(0, x)

def calculate_network(x1, x2, expected_output, w1, w2, w3, w4, w5, w6):

    H1 = relu(w1 * x1 + w3 * x2)
    H2 = relu(w4 * x1 + w2 * x2)

    O1 = w5 * H1 + w6 * H2
    squared_error = (O1 - expected_output)**2

    return(squared_error)

# (X1, X2, expected_output)
input_values = [
    (0, 0, 0),
    (0, 1, 1),
    (1, 0, 1),
    (1, 1, 0)
]

total_squared_errors = 0

for inputs in input_values:
    total_squared_errors += calculate_network(*inputs, 1, -1, 0, 1, -1, 1)

mse = total_squared_errors/(len(input_values))
print(f"Mean Squared Error: {mse}")
```

```
Mean Squared Error: 0.75
```

Part c)

In [26]:
```
from itertools import product

def relu(x):
    return max(0, x)

def calculate_network(x1, x2, expected_output, w1, w2, w3, w4, w5, w6):

    H1 = relu(w1 * x1 + w3 * x2)
    H2 = relu(w4 * x1 + w2 * x2)

    O1 = w5 * H1 + w6 * H2

    squared_error = (O1 - expected_output) ** 2

    return squared_error

# (X1, X2, expected_output)
input_values = [
    (0, 0, 0),
    (0, 1, 1),
    (1, 0, 1),
    (1, 1, 0)
]

# Define possible weight values (-1, 0, 1)
possible_weights = [-1, 0, 1]
solution_counter = 1

# Iterate over all possible combinations of weights
solutions = []
for w1, w2, w3, w4, w5, w6 in product(possible_weights, repeat=6):
    total_squared_errors = 0

    for inputs in input_values:
        total_squared_errors += calculate_network(*inputs, w1, w2, w3, w4, w5, w6)

    mse = total_squared_errors / len(input_values)

    if mse == 0:
        solutions.append((w1, w2, w3, w4, w5, w6))
        print(f"Solution {solution_counter} where MSE is {mse}: (w1 = {w1}, w2 = {w2}, w3 = {w3}, w4 = {w4}, w5 = {w5}, w6 = {w6})")
        solution_counter += 1

print(f"Total number of solutions: {len(solutions)}")
```

```
Solution 1 where MSE is 0.0: (w1 = -1, w2 = -1, w3 = 1, w4 = 1, w5 = 1, w6 = 1)
Solution 2 where MSE is 0.0: (w1 = 1, w2 = 1, w3 = -1, w4 = -1, w5 = 1, w6 = 1)
Total number of solutions: 2
```

Exercise 3

Part 1

A deep learning, neural network, based model would likely yield a better churn prediction model due to the abundance of data. Additionally, given the information we have on each customer it is likely that the data is highly non-linear. However, with the advantage of accuracy and ability to handle complex non-linear data it introduces the risk of being significantly more difficult to interpret (which might not be acceptable for a business)

In fact, this is a problem that I worked on during my time at Webflow. Our customer volume was far less than the 100 million customers mentioned in this question. Due to the relatively "low" number of training data points I opted for a traditional machine learning model and spent about 2 months cleaning data and doing feature engineering to create and test thousands of features for training the model. Ultimately, the model gave us directional indicators for customer groups (aka clusters) that were likely to churn but wasn't as accurate as we would have liked. This is especially true since we needed to understand the drivers behind a customer's propensity to churn so we could adjust our core product and business strategy to reduce churn.

Part 2

1. **Data Cleaning:** We have a lot of data per customer including demographic information, purchase transactions, browsing behavior and review data. It's nearly certain that this data in its raw form is noisy and incredibly messy. First order of business would be to clean the data to get it to a usable state. Make sure that for each feature we have a consistent data type and we understand the potential values that can exist for that data point. Additionally, we should identify how to handle missing values (if at all). Lastly, it might make sense for us to normalize certain features that are severly skewed.
2. **Feature Selection and Feature Engineering:** While cleaning the data we should be able to gather a high level understanding of the features that we have at our disposal. Given we're going with the neural network approach we don't need to worry too much about feature engineering and can let the model do the heavy lifting for us (spend <20% of model development time here). For feature selection we should be mindful of highly correlated features. An example of this could be total purchase amount over a customers lifetime and the average purchase amount (i.e. if we have the number of purchases then we can calculate the total purchase amount and don't need it). A few examples of features (non-exhaustive) I'd want to include for this problem are:

- Number of purchases
- Average purchase amount
- Average items ordered per purchase
- Time since last purchase
- Time since last browsing
- Number of times the customer has logged into the website in the last X months
- Age of the customer
- Country of residence (certain countries have higher likelihood of purchasing vs. others often a direct connection to the GDP of that nation)
- Number of ratings left for purchases
- Average rating of purchases

3. **Train and Test Setup:** Like we did in the in class example, we'll want to ensure that we split the data into training and testing sets. Using a train-test split will help reduce the risk of overfitting (albeit at 100 million customers this might not be as big of a risk). Additionally, we can use K-Fold Cross Validation to get a better understanding of the performance of the model on unseen data and not just have to rely on the test data.
4. **Create the Neural Network Model:** Features will be represented as neurons in the input layer based on the selection process done in step 2. After the input layer we'll have an N-number of hidden layers based and we can use previous modeling done on the data to determine the best number of hidden layers to start with (but this will likely have to be tuned). Activation functions in the output layer would probably be a sigmoid function where we predict if a user churns or not (which would be similar to a standard output of a logistic regression model). Since this is fairly complex data in the sense that it includes various discrete points of collection we'll likely have several hidden layers.

- Hidden Layer Activation: ReLU is commonly used since it's a piecewise linear function that helps handle complexity of non-linear data
- Output Layer Activation: Sigmoid function to provide the probability a user churns or not (similar to logistic regression)

5. **Loss Function:** We have a few options here. We could use the mean squared error loss function (typically used for regression problems) but I'd likely use the log loss function since we're trying to predict if a customer churns or not.
6. **Training:** Given the size of the dataset I'd start with a relatively high learning rate where, $\eta = 0.1$, to increase the speed of training. This will help me get to a result faster and understand a "baseline" of performance. As we run the model through the training data it'll use backpropagation to update the weights and biases per the output of the loss function aiming to minimize it via gradient descent. Hyperparameters, such as batch size and learning rate, can also be tuned using a grid search methodology.
7. **Evaluation:** After many epochs we can evaluate the performance of the model against the test data to get and calculate F1 score, accuracy, precision and AUC-ROC. These metrics will help us determine the effectiveness of the model to predict a customer churning or not.

Part 3

Before diving into the strategies to reduce churn, it is important to first understand the drivers of churn. This depends on the type of model used. In a traditional machine learning model or statistical model such as logistic regression (churn vs not churn) we can use feature importance to understand which features are the most important drivers of churn. However, in a neural network model it is more complicated since we might have orders of magnitude more weights than features.

Given this is a retail business, one way to reduce churn would be to create discounts to the customers about to churn. However, this comes at the cost of impacting the financials and could decrease margins. Additionally, if "word" got out that these discounts were being offered to customers (via social media platforms) it would cause an uptick of existing customers headed towards the "churn" route in hopes of getting discounts (I've witnessed this first hand in previous companies). This is a "slippery slope" problem and often has more risks than rewards.

Instead, of a traditional discount strategy or promotional offer implementation. I'd create a strategy focused on increasing customer engagement with the product, the services offered or the buying experience. A few examples of addressing this could be to send customers recommended products based off their previous purchase history or introducing some sort of rewards program that enhances loyalty and gamifies the shopping experience. The potential upside to this approach is that customers will feel more valued which will increase the product "stickiness" keep them around longer. We can show these recommendations by traditional means like email marketing however depending on the age of the customer SMS might be a more effective means of communication. There are risks with communication fatigue where customers are sick of getting too many emails or texts so we should be strategic and thoughtful about how we communicate with customers.

Given there is an enormous customer base (100 million customers), after identifying a potential mitigation strategy I'd run an A/B test to see if the changes have a statistically significant effect on reducing churn.