COMP202: Data Structures and Algorithms
Lab 6 Replacement Take Home
Directed Acyclic Graphs as Computational Networks

Introduction

Neural networks and more broadly the *connectionist* approach to machine learning are on the rise again, for the 3^{rd} time. This revolution came about with algorithmic and computational advances which lead to *deep learning*. Deep learning and big data are being used to solve hard problems such as autonomous driving, image understanding and machine translation.

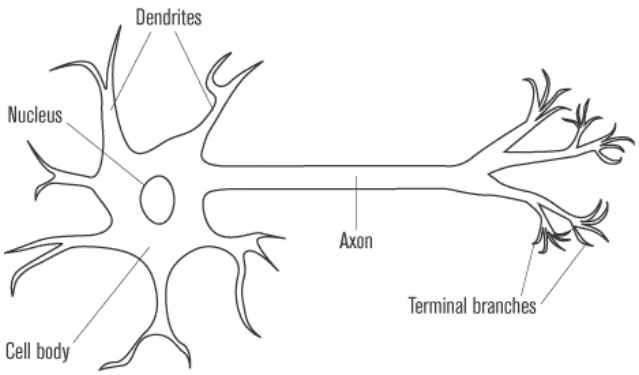
In this project, you are going to implement a generalization of *feedforward neural networks* as directed acyclic graphs, where each vertex corresponds to a neuron. For a vertex, the incoming edges carry information from previous vertices and weight of the edge correspond to connection strength. Each vertex has a transfer (or activation) function that process the incoming information. This processed information is sent via the outgoing edges to other vertices. For the sake of this project, we call the resulting graph a *computational network*.

This document will first introduce some theory. Then, it will describe what you need to do for the project. You do not need to fully understand neural networks to be able to finish the project.

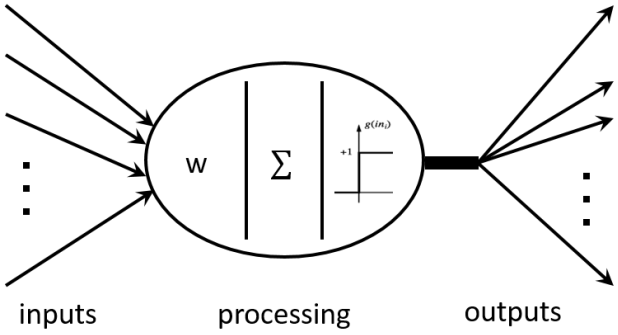
Background

A Single Artificial Neuron aka Perceptron

Artificial neural networks are inspired from biology where relatively simple processing elements are connected together to create a powerful and complex computational machinery. We start by looking at these simple processing elements. Biological neurons have 3 major parts; dendrites where they receive signals from other neurons, cell body (or soma) where these signals processed and axons where processed signals are sent to other neurons, as depicted in Fig. 1a. A neuron fires if incoming signals are above a certain threshold[1].



(a) A simplistic model of a biological neuron.



(b) A simple artificial neuron (perceptron) model.

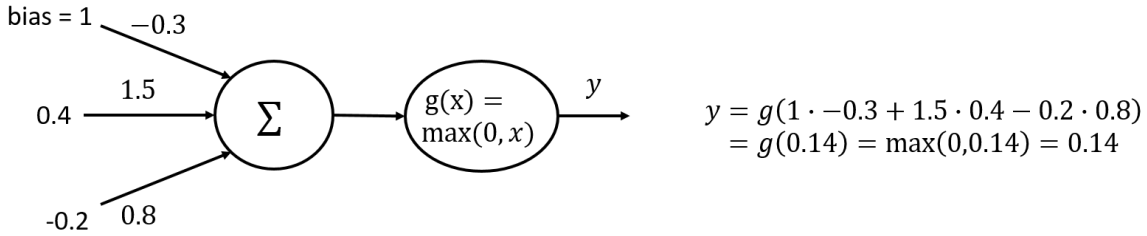
We model a single artificial neuron similar to the simplified neuron model. It has inputs (e.g. coming from other neurons), a processing stage and outputs (e.g. going to other neurons), as depicted in Fig. 1b. The inputs and outputs have weights associated with them. In the processing stage, the weighted sum of the inputs are calculated. Then, this sum is passed to a transfer function. The, output of the transfer function then becomes the output of the artificial neuron. The transfer functions are usually chosen to be similar to the threshold model of the simplified neuron. The common ones are called $sigmoid, tanh, ReLU, softplus, RBF$ etc. The details are out of scope for this project.

[1] These are highly simplified descriptions.

More formally, the input-output relation for a single artificial neuron is

$$y = g(\sum_{i=1}^{k} x_i w_i), \tag{1}$$

where x_i is the i^{th} input, w_i is the i^{th} weight, k is the number of inputs, $g(\cdot)$ is the transfer function and y is the output. This single output can be sent to multiple neurons, where it can get scaled up or down based on the weights. A crucial detail is that we also add a constant input of 1, called the *bias*, to all the neurons. Note that only the bias input is constant and not its weight! For example:



DAGs as Neural Networks

The previous section described a single artificial neuron. To have a more complex and rich computational model, multiple of these need to be connected together, just as biological neurons do. As you may have guessed, the natural way to connect them is to map the outputs of some of the neurons to the inputs of the others. This forms a directed graph, with each neuron corresponds to a vertex, each connection corresponds to an edge and the edge weight corresponds to the connection strength. Things get more complicated if we allow cycles, thus, for the sake of this project, we are going to limit these graphs to be acyclic, i.e., we are only going to be concerned with Directed Acyclic Graphs (DAG).

In the class, we have seen that there must be at least one vertex with no incoming edges and one vertex with no outgoing edges in a DAG. In our computational network, we treat the vertices with no incoming edges as *input neurons* and vertices with no outgoing edges as *output neurons*. We treat these slightly differently than other neurons. The input neurons do not have any bias input. The input to the entire network is passed to the input neurons and passed through their transfer function, and the output of the network is taken from the output neurons. As a consequence, the input dimensionality must be the same as the number of input neurons. The same logic also applies to the output dimensionality and the number of output neurons. If we let, $x = [x_1, x_2, \ldots, x_{di}]^T$ be the input vector to the network where di is the number of input neurons and $y = [y_1, y_2, \ldots, y_{do}]^T$ be the output vector where do is the number of output neurons, then we say that the give DAG represents the function $y = f(x)$. An example of a computational DAG is given in Fig. 2.

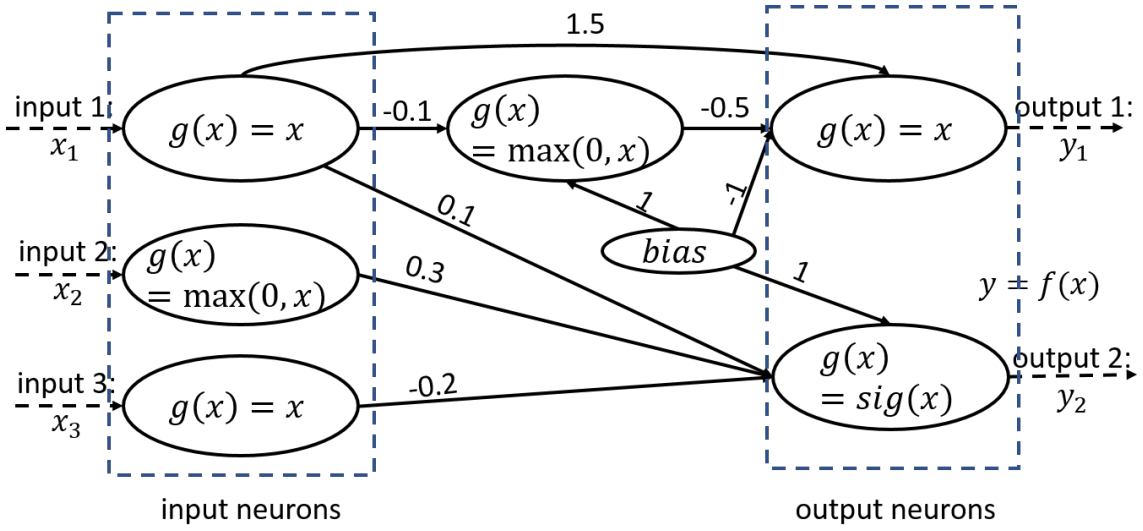


Figure 2: An example computational network.

Calculating the output given input is usually called to *forward pass*. As an exercise, let's calculate the output given $x = [2, -1, -1]^T$, by hand, where the sigmoid function is defined as $sig(x) = 1/(1 + e^{-x})$. The output should be $y = [1.6, 0.802]^T$. The steps are embedded in the table below.

Name	Incoming Sum	Output
Input 1	2 (as input to the network)	2
Input 2	-1 (as input to the network)	$max(0, -1) = 0$
Input 3	-1 (as input to the network)	-1
Intermediate	$-0.1 \cdot 1 + 1 \cdot 1 = 0.8$	$max(0, 0.8) = 0.8$
Output 1	$-0.5 \cdot 0.8 + 1.5 \cdot 2 + (-1) \cdot 1 = 1.6$	1.6 (output of the network)
Output 2	$0.1 \cdot 2 + 0.3 \cdot 0 + (-0.2) \cdot (-1) = 1.4$	$sig(1.4) = 0.802$ (output of the network)

The main purpose of neural networks is to learn a desired input-output mapping, the $f(\cdot)$ function, given data. This is done by calculating edge-weights, given input-output pairs. The algorithm to calculate weights is called *back-propagation*. Back-propagation algorithm is a *gradient descent* algorithm, where the weights are updated iteratively. We start with a random DAG. Given the desired inputs we calculate outputs. We then take the squared difference between these calculated outputs and the desired outputs and call this *error*. Given this error definition, we can derive an analytical formulation of the partial derivatives of the error with respect to the edge weights. Then, these partial derivatives are used to update the weights. After this, we go back to output calculation step and repeat until either the weights converge or we hit maximum number of iterations. The derivation and a more detailed explanation of this algorithm is out of scope of this project. However, we provide the derived equations here.
Some definitions (for a given input-output pair (x, y)):

- x: input vector and y: desired output vector
- in_i: the weighted input sum, $g_i(\cdot)$: the transfer function, and $o_i = g_i(in_i)$: the output of the vertex i
- $f_i(x)$: the calculated output, y_i: the desired output, and $e_i = 1/2(y_i - f_i(x))^2$: the error for the output vertex i
- $e = \sum_{i=1}^{n} e_i$: the total error of the network with vertices from 1 to n
- w_{ji}: the weight of the directed edge between vertices j and i
- $\delta_i = \delta e / \delta x_i$: partial derivative of the total error with respect to the incoming sum for vertex i
- V_i: outgoing neighbors and U_i: incoming neighbors of vertex i
- α: Learning rate

Then, for a given input-output pair, we have the following equations.
Forward pass:

$$o_i = \begin{cases} g_i(x_i) & \text{if the vertex } i \text{ is an input unit} \\ g_i(\sum_{k \in U_i} o_k w_{ki}) & \text{otherwise} \end{cases} \tag{2}$$

Backward pass:

$$\delta_i = \begin{cases} (y_i - f_i(x)) \frac{dg_i(in_i)}{din_i} & \text{if the vertex } i \text{ is an output unit} \\ \sum_{k \in V_i} \delta_k w_{ik} \frac{dg_i(in_i)}{din_i} & \text{otherwise} \end{cases} \tag{3}$$

Weight update:

$$w_{ji} = w_{ji} + \alpha \frac{\partial e}{\partial w_{ji}} \quad \text{where} \frac{\partial e}{\partial w_{ji}} = \delta_i o_j \tag{4}$$

For training with multiple input-output pairs, you either update the weights as you go over each pair or calculate a cumulative weight change ($\Delta w_{ji} = \sum_d \frac{\partial e^d}{\partial w_{ji}}$, where d represents the pairs) and then update the weights after going over each pair as $w_{ji} = w_{ji} + \alpha \Delta w_{ji}$. Note that there are other methods to do back-propagation but these are out of our scope.

Project

We are going to ask you to implement certain methods, detailed below. The input/output will be handled by our code. If you implement the methods correctly, you will get the correct output. You will not need to change anything in the `main` function. This project will be graded based on output matching. This implies that **we are not going to go into your code** to give you partial credits! Instead we are going to check how much of your output matches with the correct one. On the flip side, this gives you flexibility to go wild with your implementation.

Example outputs will be given to you so that you can compare your implementation. **Note that the given code will write the outputs to a file and not on your screen!** You can then compare the given file and the file generated by your implementation. Note that due to floating point arithmetic and potentially different machine precision, some of the outputs might not match (e.g. error values after training, network outputs) 100%. Furthermore, different implementations of the same algorithm might create different but equaly legitimate results. We are going to grade you with these in mind.

Important: If you think that it will be easier for you to implement everything from scratch, you are free to do so. You need to adhere to the output specifications which are not explicitly detailed here but implicit in the code. Should you chose this path, you will potentially want to copy over the input and output functions and perhaps modify them to fit your code or understand what they do.

For the remainder of the document, you should open the accompanying source code and look at the comments for more detail. This section will first introduce the classes that you are going to use and/or modify. Then it will give more details on what you are expected to do. **Helpful Note:** You are free and encouraged to create your own methods and/or fields to help you. You can modify any existing method and are even free to change method/function signatures as long as you get the correct implementation!

The Vertex and The Edge Classes

As mentioned above, we treat each neuron as a vertex in a DAG. We no longer have a `VertexType` since our vertex contents are already known. However, our vertex class contains additional fields and methods. Specifically;

Class `Vertex`	
Field/Method	Description
`String name`	Name of the vertex, to be used in accessing (e.g. within a `HashMap`).
`double incomingSum`	Variable to keep the incoming weighted sum. Will be passed through the transfer function to get the output (x_i).
`Activation f`	Class representing the activation function. Look through the code and the `Activation` interface for more details ($g_i(\cdot)$).
`double delta`	Variable to keep the partial derivative of the error with respect to the `incomingSum`. (δ_i).
`double getOutput()`	Method to pass the incoming sum through the transfer function ($o_i = g_i(x_i)$).
`double getBackDeriv()`	Method to pass the incoming sum through the derivative of the transfer function ($dg_i(x_i)/dx_i$).

The edge class does not have anything new so we are going to skip it here but you can take a look at the code. Your main interaction with the edge class will be with its weight.

The Graph Class

The graph class has some changes and additions. It no longer extends an abstract base class but is self-contained. Some functions are removed. You are going to only need to modify Graph.java file but will also need to look at other files.

This class has the methods for the forward pass (calculating the output of the network) and the training (backward pass and weight updates) that you are going to implement, among other methods. For all the other methods, their names, comments and the code should be self-explanatory.

The inputs and outputs are represented by hashmaps. The keys correspond to vertex names and the values correspond to either their input or output. As a result, the forward pass accepts inputs as a hashmap and outputs another hashmap. For training input-output pairs are stored in separate lists of hashmaps. We are going to make sure that they match.

Things that are of interest in the new graph class are given in the table below.

Class `Graph`	
Field/Method	Description
`HashSet<Vertex> inputVertices`	This set contains the input vertices, i.e., vertices with no incoming edges. Filled using the `updateInputOutputVertices` method.
`HashSet<Vertex> outputVertices`	This set contains the output vertices, i.e., vertices with no outgoing edges. Filled using the `updateInputOutputVertices` method.
`Vertex addVertex(String name, String activation)`	Adds a vertex to the graph with the given name and the activation function. Returns the newly added vertex.
`int neighborInfo(Vertex v, Vertex u)`	Checks whether two vertices are neighbors. Returns 1 if `v` has an outgoing edge to `u`, -1 if vice-versa and 0 if no edge between them exists.
`void refresh()`	Reset the internal variables of all the vertices so that next calculations are correct.
`List<Vertex> topologicallySortedVertices`	The list of topologically sorted vertices, calculated in `checkAndInitialize`
`List<Vertex> reverseTopologicallySortedVertices`	The list of reverse topologically sorted vertices, calculated in `checkAndInitialize`
`boolean checkAndInitialize()`	After creating the graph, figure out which vertices are inputs and which vertices are outputs, add a bias vertex as input to all vertices but the input ones, topologically sort the vertices and check for cycles. Return the result of cycle check
`ArrayList<Vertex> topologicalSort()`	The topological sort algorithm. Returns the list of topologically sorted vertices or `null` if the graph is cyclic. **You need to implement this, details below.**
`Edge addEdge(String startVertexName, String endVertexName, double weight)`	Add a new edge between two given vertices with the given weight. Returns the newly added edge. **You need to implement this, details below.**
`HashMap<String, Double> calculate(HashMap<String, Double> inputs)`	Given the inputs, runs a forward pass through the network and returns the output **You need to implement this, details below.**
`double train(List<HashMap<String, Double>> iData, List<HashMap<String, Double>> oData, double alpha, double epsilon, int maxIter)`	Given the input-output pairs, train the network with batch updates. Calls the `singleBackwardPass` method. Returns the final training error.
`double singleBackwardPass(HashMap<String, Double> inputs, HashMap<String, Double> outputs, HashMap<Edge,Double> cumulativeUpdate)`	Calculates the weight updates for a single input-output pair and adds these to the `cumulativeUpdate` to be used after all data points are done in the current iteration. Returns the training error for a single input-output pair. **You need to implement this, details below.**
`void printMap(Map<?, ?> m)`	Utility function to print a map (e.g. a `HashMap`), useful for debugging.
`void printList(List<?> l)`	Utility function to print a list (e.g. an `ArrayList`), useful for debugging.

Building and Checking the Graph: `addEdge` and `topologicalSort`

The `Edge addEdge(String startVertexName, String endVertexName, double weight)` method takes two vertex names, first one being the source vertex and the second one being the target vertex, and the edge weight between these vertices. If succesful, it returns the a valid edge or `null` otherwise. You need to implement this yourself. Things you need to do in this method:

- If the vertex names are the same return `null`. **DO NOT** print anything out or you might mess up auto-grading.
- If a vertex with any of the supplied names is not in the graph, add it to the graph with the supplied name. Take a look at what the `addVertex` method returns.
- If there is already an edge between the given vertices in the correct direction, update its weight. Take a look at what the `neighborInfo` method returns.
- If there is already an edge between the given vertices but in the wrong direction, return `null`. **DO NOT** print anything out or you might mess up auto-grading. Take a look at what the `neighborInfo` method returns.
- If there is no edge between the vertices, create a new edge. Make sure to update all necessary containers (Hint: There are 3 of them).
- Finally, return the valid edge, either the newly created one or the updated one, if there were no issues.

After a file is processed, the `topologicalSort` method is called to check if we end up with a DAG or not. Implementing this shoud be fairly straight-forward. The slides have the DFS based algorithm for topological sort. You need to pay attention to what you return. Topologically sorted vertices will be used in calculating the network output and training.

Forward Pass: `calculate`

The `HashMap<String, Double> calculate(HashMap<String, Double> inputs)` method calculates the output of the network given an input vector. The input vector is represented by a hashmap where the keys are the vertex names and the values are the inputs. You need to implement this yourself, based on the Eq. 2. You should also look at the definitions. Things you need to do in this method:

- Make sure that the input vector is correctly mapped to the `incomingSum` field of the input vertices (see `inputVertices`).
- Go over the remaining vertices and update their `incomingSum`. Make sure you use the edge weights and the `getOutput` method to get the outputs of the relevant vertices.
- Finally, create another hashmap representing the output of the network and fill it with the outputs of the output vertices. (see `outputVertices`) Return this hashmap.

Note that you should not call `getOutput` on any vertex that does not have its `incomingSum` fully computed. To do this, you need to make sure that its incoming neighbors also have their sums computed (seeing any pattern?). If you go over the vertices in a topological order you do not need to explicitly worry about this. I leave it upto you to figure out how to integrate everything.

Backward/Train: `singleBackwardPass`

The `double singleBackwardPass(HashMap<String, Double> inputs, HashMap<String, Double> outputs, HashMap<Edge, Double> cumulativeUpdate)` method calculates for a given input-output pair and accumulates these updates and returns the the training error for the pair. Note that the actual update is done in the `train` method. You need to implement this yourself, based on the Eq. 3 and the Eq. 4. You should also look at the definitions such as the error. Things you need to do in this method:

- Do a forward pass but do not call `refresh` afterwards. This will give you the network output and will fill the `incomingSum` fields for the vertices so that you can call `getOutput` and `getBackDeriv` whenever necessary.
- Starting from the output vertices, calculate the `delta` field for all the vertices. Note that equations for output vertices (see `outputVertices`) are different than other vertices. Make sure you use the edge weights and the `getBackDeriv` method of the relevant vertices.

- Also calculate the squared training error for the given input-output pair. Do not forget the 1/2 part. Recall that for multiple outputs, you need to add their squared error together.
- After this step, you need to go over all the edges, and calculate their update. Pay attention to Eq. 4, since the deltas are not enough. Note that you need to accumulate these updates in the `cumulativeUpdate` hashmap. Do not overwrite existing updates!
- Finally, return the squared training error.

Note that to calculate the `delta` of any vertex, you need to make sure that its outgoing neighbors have their `delta`s calculated (seeing any pattern?), similar to the forward pass case. If you go over the vertices in a reverse topological order you do not need to explicitly worry about this. I leave it upto you to figure out how to integrate everything.

Shameless Plug

If the topics of lab 4 (text processing), replaced lab 6 (image processing) and this take home (machine learning) excites you, I highly recommend that you take the artificial intelligence class that I teach, the new machine learning class that me, Deniz hoca and Mehmet hoca (from industrial engineering) designed, which will be taught by Mehmet hoca, and the machine vision class that Yucel Hoca teaches. These classes will be offered in the Fall 2017 semester. You can further follow these up by the deep learning class that Deniz hoca will teach in the Spring 2018 semester.