Problem 10

Consider the simple experts setting: we have n experts $E_1, \ldots, E_n$, and each one makes a 0/1 prediction each morning. Using these predictions, we need to make a prediction each morning, and at the end of the day we get a loss of 0 if we predicted right, and 1 if we made a mistake. This goes on for T days.

Consider an algorithm that at every step, goes with the prediction of the 'best' (i.e., the one with the least mistakes so far) expert so far. Suppose that ties are broken by picking the expert with a smaller index. Give an example in which this strategy can be really bad – specifically, the number of mistakes made by the algorithm is roughly a factor n worse than that of the best expert in hindsight.

We will show that given n experts and this algorithm, we can construct an adversarial scenario where the algorithm makes a factor of n more mistakes than the best expert. Consider the scenario where the true label is 0 for all T days. Each experts prediction is based on the following function of the current day t

$$prediction_i(t) = \begin{cases} 1 & t \ mod \ n = i - 1 \\ 0 & otherwise \end{cases}$$

In this scenario the algorithm will make T mistakes (always predicts wrong) and the best expert will be wrong T/n number of times.

To give a concrete example, consider the scenario where we have 3 experts and we run the algorithm for 9 days.

expert / t	0	1	2	3	4	5	6	7	8
E_1	1	0	0	1	0	0	1	0	0
E_2	0	1	0	0	1	0	0	1	0
E_3	0	0	1	0	0	1	0	0	1
true prediction	0	0	0	0	0	0	0	0	0
selected expert	E_1	E_2	E_3	E_1	E_2	E_3	E_1	E_2	E_3

We can see that in this example the algorithm always makes a mistake as it will cycle through the experts and they each will predict 1. Each expert is tied for being the best expert as they are each wrong T/n or 3 times. We have shown that this algorithm can make a factor of n more mistakes than the best expert.

Problem 11

We saw in class a proof that the VC dimension of the class of n-node, m-edge *threshold* neural networks is $O((m+n)\log n)$. Let us give a "counting" proof, assuming the weights are binary (0/1). (This is often the power given by VC dimension based proofs – they can 'handle' continuous parameters that cause problems for counting arguments).

(a) Specifically, how many "network layouts" can there be with n nodes and m edges? Show that

$\binom{n(n-1)/2}{m}$ is an upper bound.

From the text, we know a feedforward neural network is defined as a directed acyclic graph. It's known that the maximum number of edges a DAG can contain is $n(n-1)/2$. To see this, the first node can point to all other nodes but itself, the next can point to all nodes other than itself and the first node and so on.

$$(n-1) + (n-2) + ... + 2 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Given the maximum number of edges a network can contain is $n(n-1)/2$, we can observe that there exists at most $\binom{n(n-1)/2}{m}$ network layouts with n nodes and m edges. Each distinct network layout is defined by *choosing* m edges from a maximum possible $n(n-1)/2$ edges. Thus, the binomial coefficient captures all possible network layouts and is an upper bound.

(b) Given a network layout, argue that the number of 'possible networks' is at most $2^m(n+1)^n$. [HINT: what can you say about the potential values for the thresholds?]

Given a network layout, the number of 'possible networks' is defined by the number of possible threshold and edge weight combinations. Consider a feedforward neural network with n nodes and $m = n(n-1)/2$ edges, which is the maximum number of edges a DAG can contain (as described in part a).

From the problem, we are assuming the weights are binary (0/1). It follows that given m edges, that can each take a value from $\{0, 1\}$, there are 2^m possible ways to select these weights.

The network is constructed such that node n_i has an in-degree of $(n-i)$ (from the proof of constructing a DAG with the maximal number of edges). As such, node n_i has a potential threshold value of $(n-i)$ as the edge weights can take a value from $\{0, 1\}$. This gives

$$(n-1) * (n-2) + ... + 2 + 1 = \prod_{i=1}^{n-1} i = (n-1)!$$

potential values for the thresholds. It is clear that $(n-1)! < (n+1)^n$. Thus, the number of 'possible networks' given by the feedforward neural network with the maximal number of edges (which maximizes the range of potential values for the thresholds) is upper bounded by $2^m(n+1)^n$.

$$2^m(n-1)! \leq 2^m(n+1)^n$$

(c) Use these to show that the VC dimension of the class of binary-weight, threshold neural networks is $O((m+n)\log n)$.

claim: A finite hypothesis class $\mathcal{H}$ has a VC dimension of at most $log_2|\mathcal{H}|$.

proof by contradiction: Assume that the opposite $VC(\mathcal{C}) > log_2|C|$ is true.

Take a finite instance space X of size d. There exists 2^d (the number of possible binary vectors of length d) ways to partition X. Thus the $|\mathcal{H}| = 2^d$.

Given that X is of size d, it holds that $0 \leq VC(C) \leq d$ as $\mathcal{H}$ can shatter **at most** d points (the size of the instance space). Visiting the initial assumption:

$$VC(\mathcal{H}) > \log_2 |H|$$

$$VC(\mathcal{H}) > d$$

Arriving at the contradiction $0 \leq VC(\mathcal{H}) \leq d$ and $VC(\mathcal{H}) > d$. Thus proving $VC(\mathcal{H}) \leq log_2|\mathcal{H}|$.

From part b we showed that there are at most $2^m(n+1)^n$ possible networks for a network layout giving $|\mathcal{H}|$ is at most $2^m(n+1)^n$.

$$log_2(2^m(n+1)^n) = m + (n * log(n)) + \frac{1}{log(2)}$$

Showing the VC($\mathcal{H}$) $\leq O(m + nlog(n))$. Thus we can see $VC(\mathcal{H})$ is upper-bounded by $O((m + n)\log n)$.

Problem 12 - Importance of Random Initialization

Consider a neural network consisting of (resp.) the input layer x, hidden layer y, hidden layer z, followed by the output node f. Suppose that all the nodes in all the layers compute a 'standard' sigmoid. Also suppose that every node in a layer is connected to every node in the next layer (i.e., each layer is fully connected).

Now suppose that all the weights are initialized to 0, and suppose we start performing SGD using backprop, with a fixed learning rate. Show that at every time step, all the edge weights in a layer are equal.

We suppose that all the weights $\mathbf{w}$ are initialized to 0 before SGD is performed on the neural network. We will show that at every time step, all of the edge weights in a layer update by the same quantity keeping them equal. At each time step t we sample $(\mathbf{x}, \mathbf{y}) \sim \mathcal{D}$, calculate the gradient $\mathbf{v_t}$ using the backpropagation algorithm and update the weights $\mathbf{w}$.

The input to the backpropagation algorithm is the sampled example $(\mathbf{x}, \mathbf{y})$, weight vector $\mathbf{w}$, the network (V, E), and an activation function σ. In this example we have three hidden layers x, y, and z. During the *forward step* of the backprop we set $\mathbf{o_0} = \mathbf{x}$ and calculate $a_{t,i}$ and $o_{t,i}$ in order where t is the current hidden layer and i is the node in that layer. The values for $a_{t,i}$ and $o_{t,i}$ are calculated as such:

$$a_{t,i} = \sum_{j=1}^{k_{t-1}} \mathbf{w}_{t-1,i,j} * o_{t-1,j}$$

$$o_{t,i} = \sigma(a_{t,i})$$

The important observation here is all the weights $\mathbf{w}_{t-1,i,j}$ are zero causing all $a_{t,i}$ to be zero. As such, all $o_{t,i}$ (for a given t or layer) with be equal.

During the *backward step* these $o_{t,i}$ are used to calculate the partial derivatives that define the gradient

$$\delta_{t,i} = \sum_{j=1}^{k_{t+1}} \mathbf{w}_{t,j,i} * \delta_{t+1,j} * \sigma'(a_{t+1,j})$$

As such during the update step in SGD using the $\mathbf{v_i}$ gradients produced by backprop

$$\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta(\mathbf{v_i} + \lambda \mathbf{w^{(i)}})$$

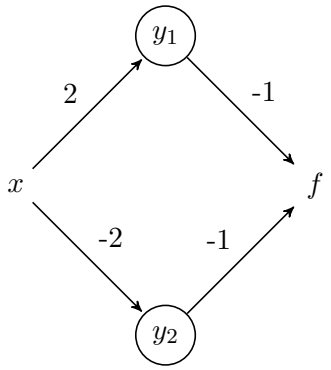
Will produce identical updates to the weights for each node in a layer. Thus all the edge weights in a layer stay equal at each time step t.

Problem 13

Let us consider networks in which each node computes a rectified linear (ReLU) function (described in the next problem), and show how they can compute very 'spiky' functions of the input variables. For this exercise, we restrict to one-variable.

(a) Consider a single (real valued) input x. Show how to compute a "triangle wave" using one hidden layer (constant number of nodes) connected to the input, followed by one output f. Formally, we should have $f(x) = 0$ for $x \leq 0$, $f(x) = 2x$ for $0 \leq x \leq 1/2$, $f(x) = 2(1 - x)$ for $1/2 \leq x \leq 1$, and $f(x) = 0$ for $x \geq 1$. [HINT: choose the thresholds and coefficients for the ReLU's appropriately.] [HINT2: play with a few ReLU networks, and try to plot the output as a function of the input.]

After some experimentation, I came up with a neural network with one hidden layer of size two that computes the given triangle wave function.



Where y_1 and y_2 are threshold functions that take in the input weighted defined as such

$$y_1(z) = \begin{cases} z-1 & z > 1 \\ 0 & otherwise \end{cases}$$

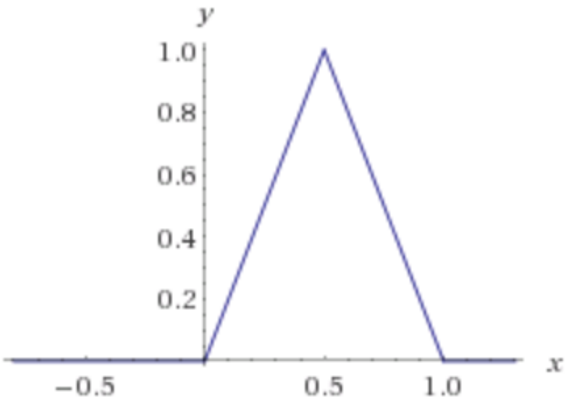
$$y_2(z) = \begin{cases} z+1 & z > -1 \\ 0 & otherwise \end{cases}$$

and the *output* function f is also a threshold function

$$f(z_1, z_2) = \begin{cases} z_1 + z_2 + 1 & z_1 + z_2 > -1 \\ 0 & otherwise \end{cases}$$

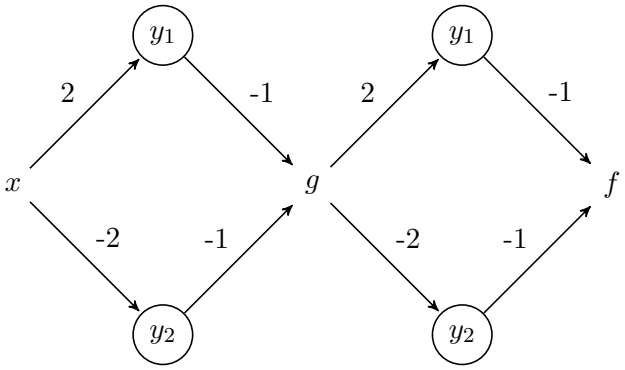
This network is defining a composition of 3 ReLU functions, which as a function of x can be expressed as such

$$f(x) = max(0, -max(0, 2x-1) - max(0, -2x+1) + 1)$$



(b) What happens if you stack the network on top of itself? (Describe the function obtained). [Formally, this means the output of the network you constructed above is fed as the input to an identical network, and we are interested in the final output function.]

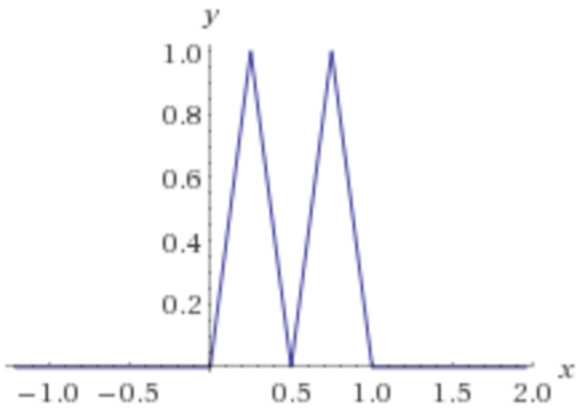
Taking the network from part a, we can stack the network on top of itself where y_1, y_2, and f are defined as they are defined in part a



We can define $f(x)$ as a composition of $g(x)$ where $g(x)$ is the output of the initial network and becomes input for the identical second network

$$g(x) = max(0, -max(0, 2x - 1) - max(0, -2x + 1) + 1)$$

$$f(x) = max(0, -max(0, 2g(x) - 1) - max(0, -2g(x) + 1) + 1)$$



(c) Prove that there is a ReLU network with one input variable x, $2k + O(1)$ layers, all coefficients and thresholds being constants, that computes a function that has 2^k "peaks" in the interval $[0, 1]$.

Using induction we will prove that as the number of layers in the ReLU network grows linearly the number of "peaks" in the interval $[0, 1]$ grows exponentially.

base case: In *part a* we constructed a ReLU network that had a single peak and O(1) layers ($k = 0$ and as such $2(0) + O(1)$ layers and $2^0 = 1$ peak).

inductive hypothesis: For all ReLU networks with $2k + O(1)$ layers, the network contains 2^k peaks.

inductive step: We will use the hypothesis that a ReLU network with $2k + O(1)$ layers contains 2^k peaks implies that a ReLU network with $2(k + 1) + O(1)$ layers contains 2^{k+1} peaks.

Each peak k in the network has a value of 1 which can be observed by graphing the triangle wave function. Each time we add another copy of the network on top of itself (as seen in part b) the output of the previous "network" becomes the input for the new "network". We can observe an input of 1 to the network produces an output of 0.

$$f(x) = max(0, -max(0, 2x - 1) - max(0, -2x + 1) + 1)$$

As such each k peak in the network becomes a valley when a new network is added to the overall network. Additionally, each k peak outputs the value 0.5 twice which will produce two new peaks in the overall network. The overall network gains 2 layers each time we add an additional network stack and the number of peaks doubles.

Thus when a network contains $2(k + 1) + O(1)$ layers, it will contain 2^{k+1} peaks.

Problem 14

In this exercise, we make a simple observation that width isn't as "necessary" as depth. Consider a network in which each node computes a rectified linear (ReLU) unit – specifically the function at each node is of the form $max\{0, a_1y_1 + a_2y_2 + ... + a_my_m + b\}$, for a node that has inputs $y1, ... , ym$. Note that different nodes could have different coefficients and offsets (b above is called the offset).

Consider a network with one (real valued) input x, connected to n nodes in a hidden layer, which are in turn connected to the output node, denoted f. Show that one can construct a depth $n + O(1)$ network, with just 3 nodes in each layer, to compute the same f. [HINT: three nodes allow you to "carry over" the input; ReLU's are important for this.]

For ease of explanation I will describe the equivalent networks with a concrete example of $n = 3$ (we will be able to see how this generalizes for all n). First let's visualize a network with one (real valued) input x, connected to 3 nodes in a hidden layer, which are connected to a final output node f.

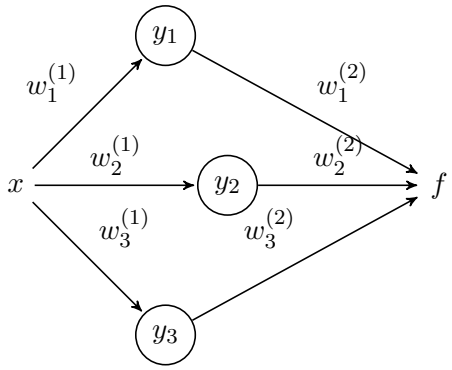


Figure 1: A network with n = 3 nodes in a hidden layer

Where y_1, y_2, y_3, and f are defined as such

$$y_1(x) = max\{0, w_1^{(1)}x + b_1\} \quad y_2(x) = max\{0, w_2^{(1)}x + b_2\} \quad y_3(x) = max\{0, w_3^{(1)}x + b_3\}$$

$$f(y_1, y_2, y_3) = max\{0, w_1^{(2)}y_1 + w_2^{(2)}y_2 + w_3^{(3)}y_3 + b\}$$

We will show that an identical network can be formed with a depth of n + O(1) and with 3 nodes in each layer where $n = 3$ for this example.

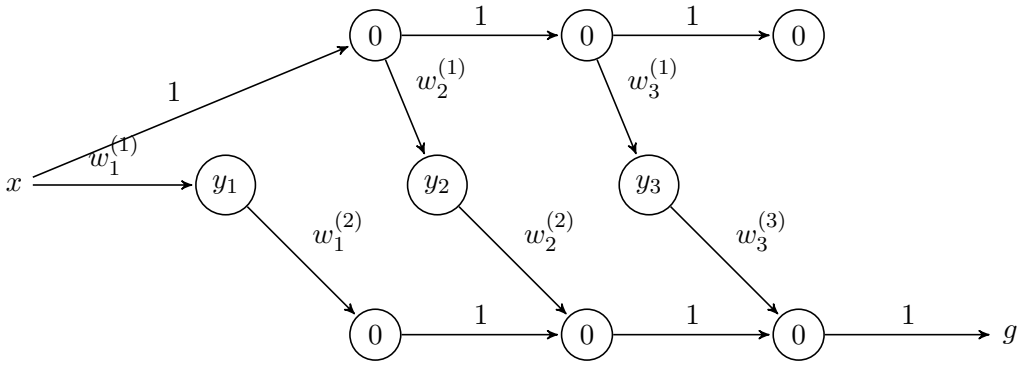


Figure 2: An identical network with a depth of 3 + O(1) with 3 nodes at each layer.

Where the input, weights, and functions are defined the same as they are in the original network. A node denoted with a 0 indicates a threshold that passes on the input regardless of it's value. As such, the top node in each layer "carries over" the initial input x. The middle node in each layer applies the functions y_1, y_2, and y_3 to the initial input x (similar to the initial networks single hidden layer). The bottom node in each layer keeps a "running sum" of each of the previous layers. Finally the sum of the n layers is fed into a final ReLU g that applies a bias b (the same bias from f in the original network) to it's single input. This produces an identical output as the original network.

Collaboration

I collaborated with Sierra Allred, Maks Cegielski-Johnson, and Dietrich Geisler on problem 11, problem 12, and problem 13.