Homework 3

Christopher Mertin
CS6966: Theory of Machine Learning

April 17, 2017

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. Given an example in whcih this strategy can be really bad – specifically, the number of mistakes made by the algorithm is roughly a factor of n worse than that of the best expert in hindsight.

    Solution:

    Consider the case where there are only two experts, $\{E_1, E_2\}$ such that E_1 predicts "0" on even days and "1" on odd, and E_2 is the opposite in such that it predicts "1" on even days and "0" on odd.

    Imagine we have the scenario where the "true value" is 1 if $t \in$ even and 0 if $t \in$ odd. For a simplified example, we can set $T = 4$ and build a table of the values to visualize the iterations. In this, $L(E_i)$ represents the *total loss* at that iteration for expert i, and $f(t)$ represents the true value.

t	E_1	E_2	$L(E_1)$	$L(E_2)$	$f(t)$
1	1		1		0
2		0	1	1	1
3	1		2	1	0
4		0	2	2	1

    This can go on to infinity, but we can see that the total loss can be computed asd $\sum_i L(E_i) = \mathcal{O}\left(n \min_i L(E_i)\right)$, meaning that for n experts, we're bounded by n times the "best expert." This can be easily seen as each of the experts will have the same value, so with n experts this goes to be n times the loss of the best.

    The above example can be extrapolated easily into n experts by simply stating that E_i predicts the same as E_1 if $i \in$ odd and E_i predicts the same as E_2 if $i \in$ even.

11. We saw in class a proof that the VC dimension of the class of n-node, m-edge *threshold* neural networks is $\mathcal{O}\left((m+n)\log(n)\right)$. 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.

Solution:

We can say that the total number of layouts based on the edges is m. We can also say that we can have 1 node with all of the m edges, or we can have up to $n-1$ different combinations with each node getting different number of edges. Therefore, we get the total number of node combinations to be

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

Using the *binomial coefficient* to get the total number of combinations gives

$$N = \binom{n(n-1)/2}{m}$$

$$N = \frac{(n(n-1)/2)!}{m!(m-n(n-1)/2)!}$$

with N being the total number of combinations of network layouts.

(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?]

Solution:

As it's a threshold neural network, we know that the weights are binary and so therer are 2^m different combinations of the weights. We can also change the threshold of the nodes to be valid/non-valid. Valid means that it is set to some useful value, non-valid means that the threshold is set to ∞ and thus the node cannot be activated. This gives $\sum_{k=0}^{n} \binom{n}{k} n^k = (n+1)^n$ different combinations of the thresholds for the nodes. We can multiply the two values together to get the total number of "possible networks," giving

$$2^m(n+1)^n$$

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

Solution:

We can say that

2

$$\begin{aligned} \text{VC Dim}(H) &\leq \log_2 |H| \\ &\leq \log_2 \left[2^m (n+1)^n\right] \\ &\leq (m+n)\log(n+1) \end{aligned}$$

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.

Solution:

The equation for the sigmoid function is

$$\sigma(\boldsymbol{\theta}_i) = \frac{1}{1 + e^{-\boldsymbol{\theta}_i}}$$

And we can obtain the derivative as

$$\sigma'(\boldsymbol{\theta}_i) = \frac{1}{[1 + e^{\boldsymbol{\theta}_i}][1 + e^{-\boldsymbol{\theta}_i}]}$$

Finally, we can say that the Jacobian/Cost function for the update is defined as

$$J_{\boldsymbol{\theta}}(\sigma) = \mathrm{diag}(\sigma'(\boldsymbol{\theta}))$$

Where the update is performed by multiplying the weight matrix by $J_{\boldsymbol{\theta}}(\sigma)$, which would obtain all of the same values in the back propogation step. There is no source of asymmetry between the neurons when they're initialized the same meaning the neural network can't learn.

It's important to note that as all the values are the same, the gradients for each of the independent weights during the back propogation step would also be the same as they all would be starting at the same value and would all have the same gradients.

4

13. Let us consider networks in whcih each node computes a rectified linear (ReLU) function, and show how they can comptue very "spiky" functions of the inptue variables. For this exercise, we restrict to one-variable.

   (a) Consider a single 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 for the ReLU's appropriately]
   Solution:
   This can be done with 3 ReLU neurons, all with equal edge weights of 1. The first neuron is defined as the function $\max\{0, 2x\}$ with a threshold of 0, the second is $\max\{0, 2(1-x) - 2x\}$ with a threshold of $1/2$, and the third is defined as $\max\{0, -2(1-x)\}$ with a threshold of 1. This can be shown to work in the figure below, where this was implemented in Python, where 500 random numbers in the range $(-1, 2)$ were generated and run through these ReLU neurons.

Single Layer Network
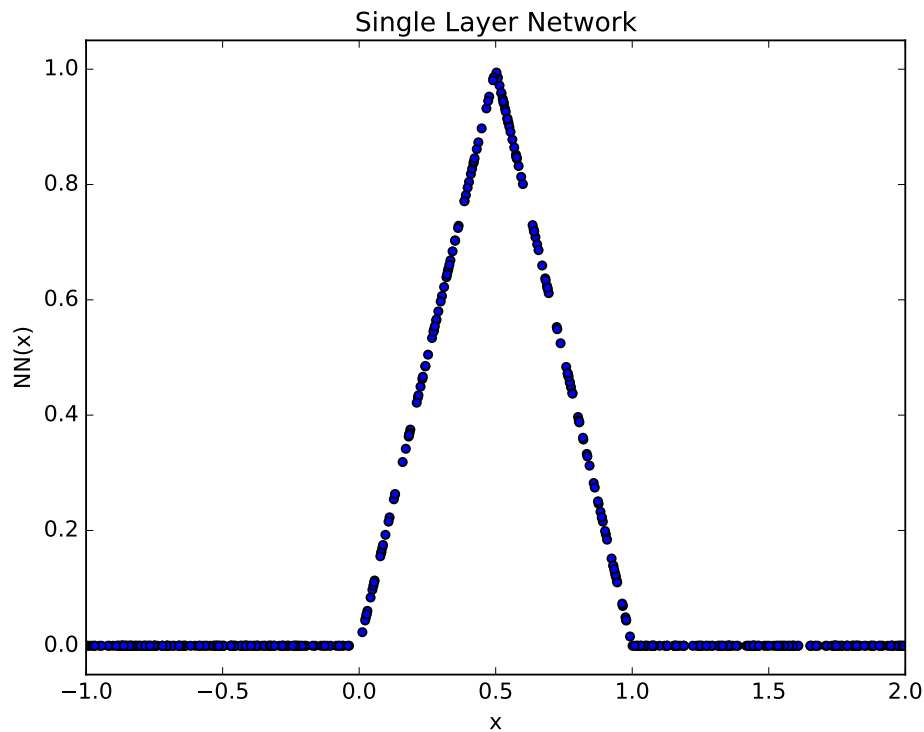
   Solution:

   (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.]

Solution:

In stacking the network ontop of itself, it creates two triangle functions, where the domain is split in half. This means that the "trough" between the two is at $x = 0.5$ with the peak of each being at 0.25 and 0.75 respectively.

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

[The function above can be shown to be impossible to approximate using a small depth ReLU network, without an exponential blow-up in the width.]

Solution:

In order to do this, you simply need to stack the network ontop of itself, giving $2k$ layers, where $\mathcal{O}(1)$ comes from the input and output layers. Each layer of the network doubles the number of peaks.

This can be seen by mapping out two layers of the network.

	x_1	x_2	x_3	x_4	x_5
x	0	1/4	1/2	3/4	1
$NN(x)$	0	1/2	1	1/2	0
$NN(NN(x))$	0	1	0	1	0

Therefore, as it doubles with every layer, we only need $2k$ layers to map out 2^k peaks.

14. In this exercise, we make a simple observatino that width isn't as "necessary" as depth. Consider a network in which each node comptues a rectified linar (ReLU) unit – specifically the function at each node is of the form $\max\{0, a_1x_1+a_2x_2+\cdots+a_mx_m+b\}$, for a node that has inputs $\{x_1, \ldots, x_m\}$. Note that different nodes could have different coefficients and offsets (b above is called the offset).

Consider a network with one input x, connect 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 $n + \mathcal{O}(1)$ network, with just 3 nodes in each layer, to compute the same f.
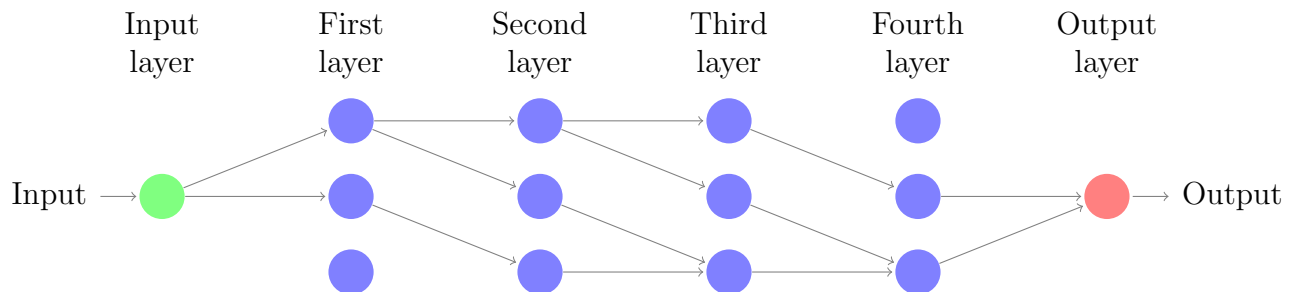
[*Hint:* Three nodes allow you to "carry over" the input; ReLU's are important for this]

Solution:

The original "wide" neural network is of the form
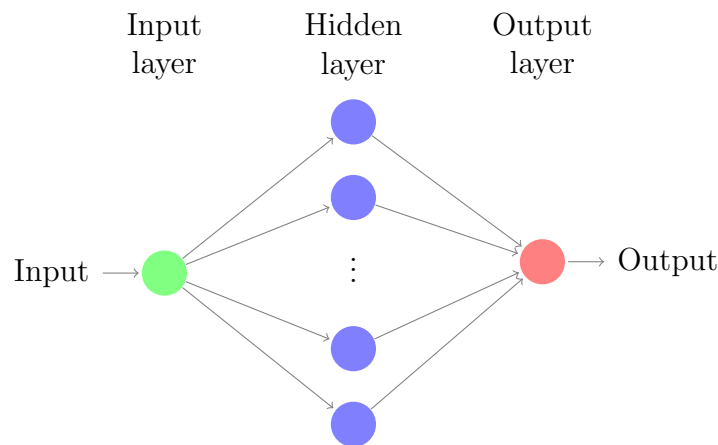
This can be changed to a "deep network" in the following way. This was made as a depth-4 network, though for a depth n network, the edges between the second and third layer can be repeated networks larger than 4.

In this network, the value of x is fed only into the first two nodes. The task of the first node is simply to pass the value of x through the network, so it will have a weight of 1. The second node in the layer calculates the weight for that node relative to x. This value is passed to the 3rd node of the *next layer* which stores the total sum of all the weights, so the weight for the third nodes will be 1 as well.