Neural Networks and Deep Learning: Exercises

Somnath Sikdar

March 1, 2020

Contents

Chapter 1

Using Neural Networks to Recognize Handwritten Digits

Exercise 1.1. Consider a network of perceptrons. Suppose that we multiply all weights and biases by a positive constant $c > 0$. Show that the behaviour of the network does not change.

Solution. First consider a single perceptron. Assume that weights and bias are $w_1, \ldots, w_n$ and b, respectively. Then $\sum_i w_i \cdot x_i + b$ and $c \cdot (\sum_i w_i \cdot x_i + b)$ have exactly the same sign and hence multiplying the weights and the bias by c will not change the behaviour of this single perceptron. Now if all perceptrons in a network have their weights and biases multiplied by $c > 0$, then each individual perceptron behaves as before and hence the network behaves as before. ■

Exercise 1.2. Suppose that we have network of perceptrons with a chosen input value $\boldsymbol{x}$. We won't need the actual input value, we just need the input to have been fixed. Suppose the weights and biases are such that all $\boldsymbol{w} \cdot \boldsymbol{x} + b \neq 0$ for the input $\boldsymbol{x}$ to any particular perceptron in the network. Now replace all the perceptrons in the network by sigmoid neurons, and multiply the weights and biases of the network by a positive constant $c > 0$. Show that in the limit as $c \to \infty$, the behaviour of this network of sigmoid neurons is exactly the same as the network of perceptrons. How can this fail when $\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$ for one of the perceptrons?

Solution. As in the previous exercise, first consider a single perceptron in the network. When this is replaced by a sigmoid neuron, and we let $c \to \infty$, $c \cdot (\boldsymbol{w} \cdot \boldsymbol{x} + b)$ tends to either $+\infty$ or $-\infty$ depending on whether $\boldsymbol{w} \cdot \boldsymbol{x} + b$ is positive or negative. The upshot is that the output of the sigmoid neuron matches that of the perceptron it replaced. Thus when every sigmoid neuron behaves as the perceptron it replaced, the network as a whole behaves similarly.

This works as long as $\boldsymbol{w} \cdot \boldsymbol{x} + b \neq 0$. If this is zero, the output of the sigmoid neuron is "stuck" at $1/2$ irrespective of the value of c, while the perceptron outputs a 0. The outputs do not match and the behaviour of the sigmoid network may be different. ■

Exercise 1.3. There is a way of determining the bitwise representation of a digit by adding an extra layer to the three-layer network given in the book. The extra layer converts the output of the previous layer in binary representation. Find a set of weights and biases for the new output layer. Assume that the first three layers of neurons are such that the correct output in the third layer (i.e., the old output layer) has activation at least 0.99, and incorrect outputs have activation less than 0.01.

Solution. Label the neurons of the third layer (the old output layer) as $0, 1, \ldots, 9$ and the neurons from the new output layer as $0', 1', 2', 3'$ with the interpretation that neuron $0'$ is the least significant bit and $3'$ is the most significant bit of the number represented by the output layer. The weight of the connection between the ith neuron from the third layer and the jth neuron of the output layer is w_{ij}, where $i \in \{0, \ldots, 9\}$ and $j \in \{0', 1', 2', 3'\}$. The bias of the jth output neuron is b_j. Denote the output of the ith neuron from the third layer as x_i. Then the input to the final layer may be represented as:

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} w_{00} & w_{10} & \ldots & w_{90} & b_0 \\ w_{01} & w_{11} & \ldots & w_{91} & b_1 \\ w_{02} & w_{12} & \ldots & w_{92} & b_2 \\ w_{03} & w_{13} & \ldots & w_{93} & b_3 \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_9 \\ 1 \end{pmatrix}$$

Now we would like z_0 to be 1 when the number is $1, 3, 5, 7, 9$ and 0 otherwise. To be able to do this, first set

$$w_{10} = w_{30} = w_{50} = w_{70} = w_{90} = +1$$

and the remaining weights of the inputs to $0'$ to -1. Set $b_0 = 0$. Now if the third layer represents $k \in \{1, 3, 5, 7, 9\}$, we would have $w_{k0} > 0.99$ and $w_{j0} < 0.01$ for all $j \neq k$. With these weights, we would have $z_0 > 0.99 - 9 \times 0.01 = 0.90$. If the third layer represents a number $k \notin \{1, 3, 5, 7, 9\}$, then $z_0 < -0.99 + 9 \times 0.01 = -0.90$. We can amplify this phenomenon by multiplying all these weights by a large positive constant. This would lead the sigmoid neuron $0'$ to output a 1 for the digits $1, 3, 5, 7, 9$ and a 0 for the remaining digits.

We can use a similar strategy for the remaining neurons of the fourth layer. For example, the second most significant bit $1'$ must be a 1 for the digits $2, 3, 6, 7, 9$ and a 0 for the remaining digits. We would then set

$$w_{21} = w_{31} = w_{61} = w_{71} = w_{91} = +1$$

and the remaining weights to -1. The bias b_1 is set to 0. ∎

Exercise 1.4. Let $C(v_1, \ldots, v_m)\colon \mathrm{R}^m \to \mathrm{R}$ be a differentiable function. Then $\Delta C \approx \nabla C \cdot \Delta \boldsymbol{v}$. Constrain $\|\Delta \boldsymbol{v}\| = \epsilon$, where $\epsilon > 0$ is a small fixed real. Show that the choice of $\Delta \boldsymbol{v}$ that minimizes $\nabla C \cdot \Delta \boldsymbol{v}$ is $\Delta \boldsymbol{v} = -\eta \nabla C$, where $\eta = \epsilon / \|\nabla C\|$.

Solution. The Cauchy-Schwarz inequality tells us that

$$\begin{aligned} |C_{v_1}^{(1)} \Delta v_1 + \cdots + C_{v_m}^{(1)} \Delta v_m| &\leq \left((C_{v_1}^{(1)})^2 + \cdots + (C_{v_m}^{(1)})^2\right)^{1/2} \left((\Delta v_1)^2 + \cdots + (\Delta v_m)^2\right)^{1/2} \\ &= \|\nabla C\| \cdot \epsilon, \end{aligned}$$

where $C_{v_i}^{(1)} = \frac{\partial C}{\partial v_i}$. Since the right-hand side is a positive number no matter what the values of the partial derivatives $C_{v_i}^{(1)}$ and the changes Δv_i in the values of the variables, the smallest possible value of the left-hand side is $-\|\nabla C\| \cdot \epsilon$. Since we are trying to minimize ΔC which is approximated by the left-hand side, the goal is to find values for the Δv_i such that minimizes the left-hand side. Observe that when we set $\Delta v_i := -\epsilon \cdot \frac{\nabla C}{\|\nabla C\|}$ for all $1 \leq i \leq m$, then the left-hand side indeed equals the said minimum value. Hence it must be that this setting of the Δv_is is the optimum. ■

Chapter 2

The Backpropagation Algorithm

2.1 The Backpropagation Equations

Before we describe anything, we briefly recap notation. We let C denote the cost function and σ the activation function of the neurons.

1. w^l_{jk} is the weight of the link between the jth neuron in layer l and the kth neuron in layer $l-1$.
2. b^l_j is the bias of neuron j in layer l.
3. z^l_j is the weighted input to neuron j in layer l.
4. $a^l_j = \sigma(z^l_j)$ is the activation of neuron j in layer l.
5. $\delta^l_j := \partial C/\partial z^l_j$ is the "error" of neuron j in layer l.

Using this notation, we may write the weighted output to neuron j in the lth layer as:

$$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j = \sum_k w^l_{jk} \sigma(z^{l-1}_k) + b^l_j,$$

where the index k runs over all neurons in layer $l-1$ and $2 \le l \le L$. Symbols such as w^l, b^l, a^l without subscripts refer to either matrices or vectors as the case may be. For example, w^l refers to the matrix whose (j,k)th element is w^l_{jk}. This matrix has as many rows as there are neurons in the lth layer and as many columns as there are neurons in layer $l-1$. The symbol b^l refers to the vector of biases b^l_j of the neurons in layer l; similarly, a^l refers to the vector of activations a^l_j of the neurons in layer l.

With this notation in hand, we may write the backpropagation equations as:

$$\boxed{\begin{aligned} \delta^L &= \nabla_{a^L} C \odot \sigma'(z^L) \\ \delta^l &= ((w^{l+1})^\intercal \delta^{l+1}) \odot \sigma'(z^l) \\ \frac{\partial C}{\partial b^l_j} &= \delta^l_j \\ \frac{\partial C}{\partial w^l_{jk}} &= a^{l-1}_k \delta^l_j \end{aligned}} \tag{2.1}$$

2.2 Backpropagation Applied to Gradient Descent

The backpropagation procedure calculates the gradient of the cost function C with respect to a single input example. To make use of backprop in the context of stochastic gradient descent, we need to take the mean of the gradient computed over all examples in a mini batch. Let's suppose that we have a mini batch with m examples $x_1, \ldots, x_m$.

1. For each training example x, set the input activation $a^1(x)$ and perform the following steps:

   (a) **Feedforward.** For $2 \le l \le L$, set $z^l(x) = w^l a^{l-1}(x) + b^l$ and $a^l(x) = \sigma(z^l(x))$.

   (b) **Output Error.** Calculate $\delta^L(x) = \nabla_{a^L} C(x) \odot \sigma'(z^L(x))$.

   (c) **Backprop.** For $L-1 \le l \le 2$, $\delta^l(x) = ((w^{l+1})^\intercal \delta^{l+1}(x)) \odot \sigma'(z^l(x))$.

   (d) **Gradients.** Calculate $\frac{\partial C}{\partial b^l_j}(x) = \delta^l_j(x)$ and $\frac{\partial C}{\partial w^l_{jk}}(x) = a^{l-1}_k \delta^l_j(x)$.

2. **Gradient Descent.** For $L \le l \le 2$, set $w^l = w^l - \frac{\eta}{m} \sum_x \delta^l(x)(a^{l-1}(x))^\intercal$ and $b^l = b^l - \frac{\eta}{m} \sum_x \delta^l(x)$.

Chapter 3

Improving the Way Neural Networks Learn

3.1 The Cross-Entropy Cost Function

Exercise 3.1. Show that the cross-entropy function is minimized when $\sigma(z) = y$ for all inputs.

Solution. The cross-entropy function is defined as:

$$C = -\frac{1}{m}\sum_{i=1}^{m}[y_i \ln(a_i) + (1-y_i)\ln(1-a_i)],$$

where the y_is are fixed and the a_is are the "variables." Now $\partial C/\partial a_i$ is given by:

$$\frac{\partial C}{\partial a_i} = \frac{1}{m}\frac{a_i - y_i}{a_i(1-a_i)}.$$

At an extremum point of C, each component of the gradient $\nabla_a C$ will be zero. This happens when $a_i = y_i$ for all $1 \leq i \leq m$.

As a side note, the function $H(y) = -[y\ln(y) + (1-y)\ln(1-y)]$ for $y \in (0,1)$ is called the binary entropy function and behaves as shown in Figure 3.1. ■

Exercise 3.2. Partial derivatives of the cross-entropy cost function in multi-layer networks.

Solution. The cross-entropy function for a single training example x for the last layer L of the network is defined as:

$$C(x) = -\sum_j \left[y_j \ln(a_j^L) + (1-y_j)\ln(1-a_j^L)\right],$$

where the sum is over all neurons j in layer L. To recap notation,

$$a_j^L = \sigma(z_j^L) = \sum_k w_{jk}^L a_k^{L-1} + b_j^L.$$

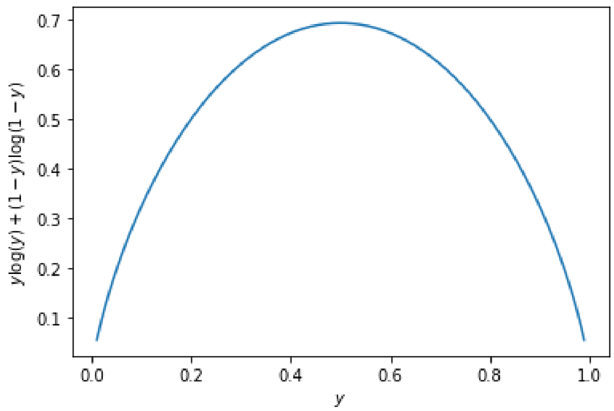

Figure 3.1: The Binary Entropy Function

For this training example x,

$$\begin{aligned}\frac{\partial C(x)}{\partial z_j^L} &= -\frac{y_j}{a_j^L}\cdot\sigma'(z_j^L) + \frac{1-y_j}{1-a_j^L}\cdot\sigma'(z_j^L)\\ &= \frac{-y_j + y_j a_j^L + a_j^L - y_j a_j^L}{a_j^L(1-a_j^L)}\cdot\sigma'(z_j^L)\\ &= a_j^L - y_j.\end{aligned}$$

The last equality follows since $\sigma'(z_j^L) = a_j^L(1-a_j^L)$.

Again, for this single training example x, $\partial C(x)/\partial w_{jk}^L$ is given by:

$$\begin{aligned}\frac{\partial C(x)}{\partial w_{jk}^L} &= \frac{\partial C}{\partial z_j^L}\cdot\frac{\partial z_j^L}{\partial w_{jk}^L}\\ &= (a_j^L - y_j)\cdot a_k^{L-1}.\end{aligned}$$

For n training examples, the cost function is defined as $\frac{1}{n}\sum_x C(x)$ and this derivative is:

$$\frac{\partial C}{\partial w_{jk}^L} = \frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j).$$

If we were to replace $C(x)$ by the usual quadratic cost $\frac{1}{2}(y_j - a_j^L)^2$, then the same derivative would have been:

$$\frac{1}{n}\sum_x a_k^{L-1}(a_j^L - y_j)\cdot\sigma'(z_j^L).$$

■

3.1.1 Deriving the Cross-Entropy Function

Given a single neuron with r input weights $w_1, \ldots, w_r$ and bias b, and a single input $\boldsymbol{x} = (x_1, \ldots, x_r)^\intercal$, we would like the cost function C to depend on the weights and the bias as follows:

$$\frac{\partial C}{\partial w_j} = x_j(a - y) \tag{3.1}$$
$$\frac{\partial C}{\partial b} = a - y, \tag{3.2}$$

where $a = \sigma(\sum_j w_j x_j + b)$ and y is the desired output corresponding to $\boldsymbol{x}$.

Using the chain rule, we obtain:

$$\begin{aligned}\frac{\partial C}{\partial b} &= \frac{\partial C}{\partial a} \cdot \frac{\partial a}{\partial z} \cdot \frac{\partial z}{\partial b} \\ &= \frac{\partial C}{\partial a} \cdot \sigma'(z) \\ &= \frac{\partial C}{\partial a} \cdot a(1-a) \end{aligned} \tag{3.3}$$

From equations (3.2 and 3.3), we obtain:

$$\frac{\partial C}{\partial a} = \frac{a-y}{a(1-a)} = \frac{1}{1-a} - y\left(\frac{1}{1-a} + \frac{1}{a}\right).$$

Integrating both sides wrt a, we obtain:

$$C = -[(1-y)\ln(1-a) + y\ln(a)] + \text{ a constant.}$$

3.2 Softmax

Consider a classification problem, where labelled examples take the form $(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x} \in \mathrm{R}^m$ and $\boldsymbol{y} \in \{0,1\}^J$ denotes to which of the J classes $\boldsymbol{x}$ belongs to. In such cases, it makes sense to have the last layer of the neural network to have J neurons with the softmax activation:

$$z_j^L = \sum_k w_{jk}^L a_k^{L-1} + b_j$$
$$a_j^L = \frac{e^{z_j^L}}{\sum_{i=1}^J e^{z_i^L}}$$

The cost associated with the input $(\boldsymbol{x}, \boldsymbol{y})$ where $y_r = 1$ is defined as the negative log-likelihood of the activation a_r^L:

$$C(\boldsymbol{x}, \boldsymbol{y}) = -\ln a_r^L.$$

The partial derivatives $\partial C/\partial b_j^L$ and $\partial C/\partial w_{jk}^L$ can be computed easily as follows. Depending on whether the index j and the class index r are the same or not, we have two cases for each partial derivative.

$$\frac{\partial C}{\partial b_r^L} = -\frac{1}{a_r^L} \cdot \left[\frac{e^{z_r^L}}{\sum_{i=1}^J e^{z_i^L}} - \left(\frac{e^{z_r^L}}{\sum_{i=1}^J e^{z_i^L}} \right)^2 \right] = -1 + a_r^L \tag{3.4}$$

$$\frac{\partial C}{\partial b_j^L} = -\frac{1}{a_r^L} \cdot \left[0 - \frac{e^{z_r^L} e^{z_j^L}}{\left(\sum_{i=1}^J e^{z_i^L}\right)^2} \right] = a_j^L. \tag{3.5}$$

The first equation is when the index $r = j$ and the second when $r \neq j$. These two expressions can be summarized into one:

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j. \tag{3.6}$$

Similarly, the partial derivative expression for $\partial C/\partial w_{jk}^L$ can be written as:

$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1} \cdot (a_j^L - y_j). \tag{3.7}$$

Exercise 3.3. Where does the "softmax" name come from? Consider the following variant of the softmax function:

$$a_j^L = \frac{e^{cz_j^L}}{\sum_{i=1}^J e^{cz_i^L}},$$

where c is a positive constant. What is the limit of a_j^L as $c \to \infty$?

Solution. Let $z_r^L = \max_i \{z_i^L\}$. We could then write the modified softmax function as:

$$a_j^L = \frac{e^{c(z_j^L - z_r^L)}}{1 + \sum_{i \neq r} e^{c(z_i^L - z_r^L)}}.$$

If $j = r$, then the numerator is 1 and the denominator approaches 1 as $c \to \infty$ and $a_j^L \to 1$. On the other hand, if $j \neq r$, the numerator $\to 0$ as $c \to \infty$; the denominator in any case approaches 1 and hence $a_j^L \to 0$. The point here is that $a_j^L = 1$ if z_j^L is the maximum and $a_j^L = 0$ otherwise. ∎

Chapter 4

Neural Networks Can Compute Any Function

This is a condensed write-up of Chapter 4 of Nielsen's book. The objective is to present the main ideas of a proof that neural networks can approximate any continuous function.

4.1 Step Functions and Function Approximation

There are two main observations in this "proof." First, that a single sigmoid neuron can approximate a step function; and second, given any interval on the real line $[a, b]$, one can construct a fixed-sized network of sigmoid neurons that takes as input a real number x and outputs a 1 if and only if $x \in (a, b)$.

It is easy to show that if we want a single sigmoid neuron to step-up from 0 to 1 at a point s on the real line, then this can be achieved by selecting a high enough weight, say $w = 1000.0$, and bias of $b = -s \times w$. This is shown in Figure 4.1.

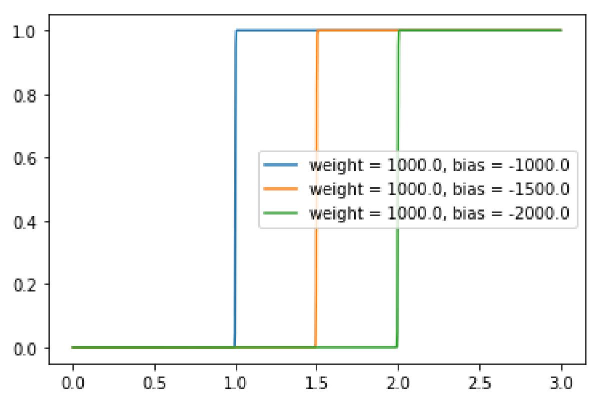

Figure 4.1: Output of a single sigmoid neuron for different values of weight and bias

We can now combine two sigmoid neurons which step up at points s_1 and s_2, with $s_1 < s_2$, to create a network that outputs a 1 if and only if $x \in (s_1, s_2)$ as shown in Figure 4.2. The top neuron of the hidden layer has a bias $b_1 = -s_1 \times w_1$, where w_1 is the weight of the link between itself and the input node. We choose w_1 to be a large enough number so that this neuron acts as a step function at the point s_1. The bottom neuron in the hidden layer has bias $b_2 = -s_2 \times w_2$, where w_2 is the weight of the link between itself and the input node. As before, we choose w_2 to be large enough so that the bottom neuron steps up at s_2.

Now the trick in making the output of the combined network to be a 1 when the input is in the interval (s_1, s_2) is by adjusting the weights of the links from the hidden layer to the output neuron. We set the weight of the upper link from the top-most neuron to the output neuron to be h and the weight of the lower link to be $-h$. Here h is just a very large number. The bias of the output neuron is set to $-h/2$. The resulting output is then given by:

$$\sigma(h \cdot a_1 - h \cdot a_2 - h/2),$$

where $a_1 = \sigma(w_1 x + b_1)$ and $a_2 = \sigma(w_2 x + b_2)$. Now $a_1 = 1$ iff $x > s_1$ and $a_2 = 1$ iff $x > s_2$. Thus if $x < s_1$, the output is $\sigma(-h/2) \approx 0$; if $x \in (s_1, s_2)$, the output is $\sigma(h/2) \approx 1$; if $x > s_2$ the output $\sigma(-h/2) \approx 0$. At the boundary points, s_1 and s_2, the output is a number between 0 and 1. This is because the neurons can only approximate a step function. This construct forms the basis of how we can approximate arbitrary continuous functions from $\mathrm{R}^m \to \mathrm{R}^n$.

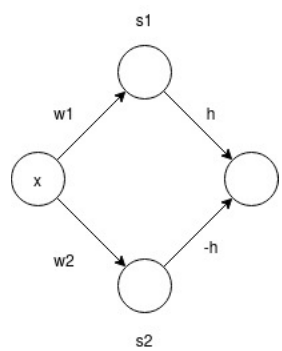

Figure 4.2: Network of sigmoid neurons to output a rectangle function.

The output of such a network is shown in Figure 4.3.

4.2 Functions of One Variable

The network described in the last section acts as an indiacator function for an interval $[s_1, s_2]$ of the real line. Consider a function $f : \mathrm{R} \to \mathrm{R}$ whose mean value in the interval $[s_1, s_2]$ is y. If we were to now weight the output of the network by y by adding an extra output node with a linear activation function, the resulting output of this new network will be a y whenever $x \in (s_1, s_2)$ and a 0 otherwise. Piecing together several of these networks would allow us to output the approximate value of f over several intervals.

In Figure 4.4, we show a network that approximates a function f over an interval $[s_1, s_5]$. In order to do this, this interval is first broken down into four sub-intervals $[s_1, s_2]$, $[s_2, s_3]$,

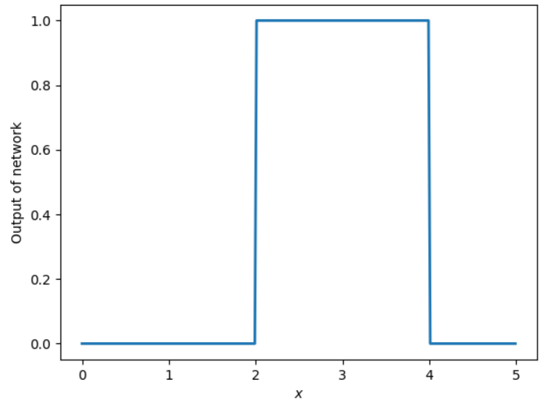

Figure 4.3: Output of the network. $w_1 = w_2 = 10000.0$ and $h = 10000.0$

$[s_3, s_4]$ and $[s_4, s_5]$. The top-most neuron from the second hidden layer from the left represents the output of the indicator function that detects whether the input x lies in the interval $[s_1, s_2]$. This neuron is labelled $I[s_1, s_2]$ to represent this fact. The output weight from this neuron is labelled $f[s_1, s_2]$ which represents the average value of the function f in the interval $[s_1, s_2]$.

The output neuron has a linear activation whereas the remaining neurons in the two hidden layers are sigmoid neurons. The network as a whole functions as follows: when $x \in [s_i, s_{i+1}]$ for $i \in \{1, 2, 3, 4\}$, the corresponding sub-network $I[s_1, s_{i+1}]$ outputs a 1; all other sub-networks output 0. The final output of the network is then $f[s_i, s_{i+1}]$, the approximate value of f in this interval. By increasing the number of intervals, one can obtain better and better approximations of f over larger intervals.

This idea is used to generalize the result to functions of several variables.

4.3 Functions of Several Variables

In order to generalize this to functions of two variables, we need to first construct step functions on the plane. We could then construct networks that represent indicator functions for rectangular regions in R^2. By taking a linear combination of such indicator functions with appropriate weights, we can approximate any continuous function $f : \mathrm{R}^2 \to \mathrm{R}^1$.

The network used to approximate a step function in R^2 is shown in Figure 4.5.

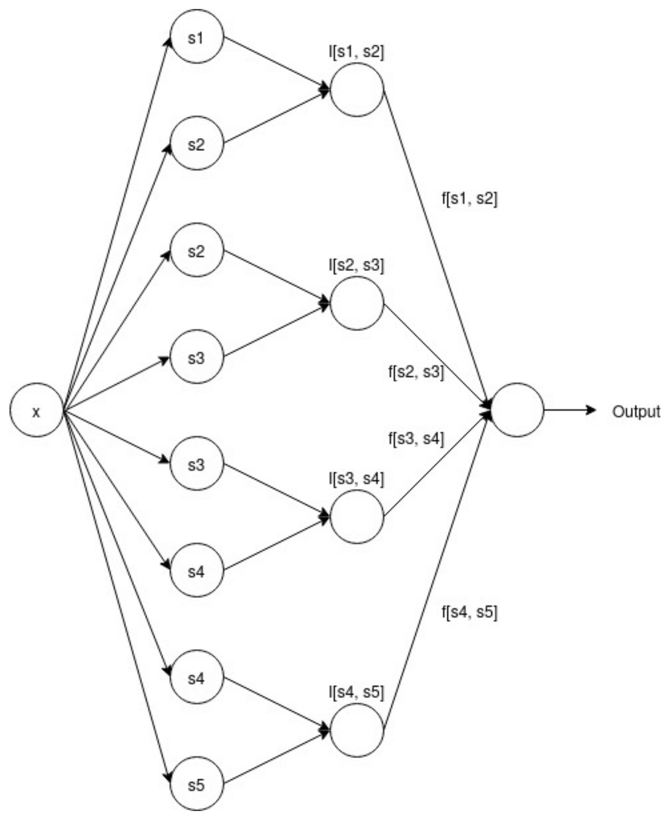

Figure 4.4: Network that approximates a function on the range $[s_1, s_5]$

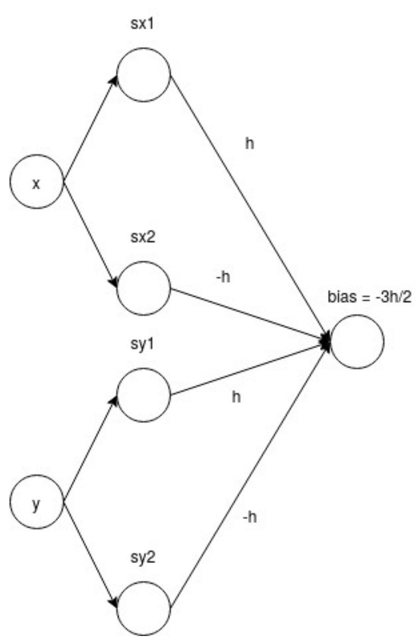

Figure 4.5: Network that approximates a tower function in two variables