Neural Networks and Deep Learning: Exercises

Somnath Sikdar

July 26, 2021

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 wont 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

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

where $C^{(1)}_{v_i} = \frac{\partial C}{\partial v_i}$. Since the right-hand side is a positive number no matter what the values of the partial derivatives $C^{(1)}_{v_i}$ 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 $2 \leq l \leq L$. 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.

To understand why it makes sense to call $\delta^l_j := \partial C/\partial z^l_j$ the "error," consider a small change Δz^l_j in z^l_j. Then the change in the cost function C as a result of this change in z^l_j is $\partial C/\partial z^l_j \cdot \Delta z^l_j$. If $\partial C/\partial z^l_j > 0$, then the cost increases as we increase Δz^l_j; thus to reduce the cost, we must choose $\Delta z^l_j < 0$. Similarly, if $\partial C/\partial z^l_j < 0$, then the cost increases if $\Delta z^l_j < 0$ and in order to reduce cost, we should select $\Delta z^l_j > 0$. Finally, if $\partial C/\partial z^l_j \approx 0$, then changes in z^l_j do not affect the final cost. Thus $\partial C/\partial z^l_j$ indicates how we should change the input z^l_j to the jth neuron in layer l.

We will derive the complete set of backpropagation equations in four steps. In what follows, we will assume that the cost function C is the usual quadratic cost.

$$C(x) = \frac{1}{2}\left\|y - a^L(x)\right\|^2 = \frac{1}{2}\sum_k (y_k - a_k^L)^2.$$

Step I. Error of the Output Layer

Let us consider the jth neuron in the last layer L. The "error" δ_j^L of this neuron is given by

$$\delta_j^L := \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial \sigma(z_j^L)}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{2.1}$$

Now $\partial C/\partial a_j^L = a_j^L - y_j$ and so $\delta_j^L = (a_j^L - y_j)\sigma'(z_j^L)$. Assuming that there are k neurons in layer L, we may write δ^L, the vector of all the errors from layer L, as follows:

$$\delta^L = \begin{pmatrix} \sigma'(z_1^L) & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma'(z_k^L) \end{pmatrix} \begin{pmatrix} \frac{\partial C}{\partial a_1^L} \\ \vdots \\ \frac{\partial C}{\partial a_k^L} \end{pmatrix} \tag{2.2}$$

Step II. δ^l in terms of δ^{l+1}

Now that we know what the error terms are in the final layer, we would like to "backpropagate" these errors from the final layer to the first layer of the network. In particular, we want to compute the errors in layer l from the errors in layer $l+1$.

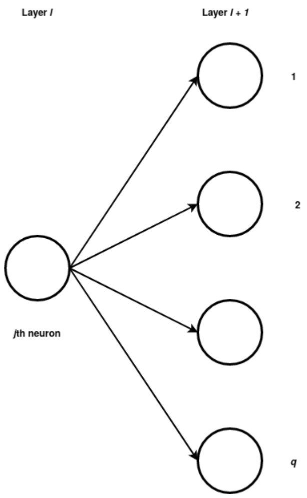
Let us assume that there are p neurons in layer l and q neurons in layer $l+1$. What we would like to show is that:

$$\delta^l = \operatorname{diag}(\sigma'(z_1^l), \dots, \sigma'(z_p^l)) \cdot (w^{l+1})^\intercal \cdot \delta^{l+1}. \tag{2.3}$$

On the righthand side of the above equation, the diagonal matrix $\operatorname{diag}(\sigma'(z_1^l), \dots, \sigma'(z_p^l))$ has order $p \times p$, the matrix w^{l+1} has order $q \times p$ and δ^{l+1} has order $q \times 1$, so that at least the order of δ^l correctly evaluates to $p \times 1$.

From (2.1), we know that the error of the jth neuron in layer l is given by $\delta_j^l = \sigma'(z_j^l) \cdot \partial C/\partial a_j^l$. The output from the jth neuron is fed to all neurons in layer $l+1$. Thus if we were to modify the input weights to the jth neuron, its output would change and that would have a cascading effect on the rest of the network. We wish to evaluate how a change in the outputs of neuron j in layer l affects the output of the neurons in layer $l+1$, assuming that all other weights in the network remain fixed.

$$\begin{aligned} \frac{\partial C}{\partial a_j^l} &= \sum_{k=1}^{q} \frac{\partial C}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial a_j^l} \\ &= \sum_{k=1}^{q} \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{\partial a_j^l}. \end{aligned} \tag{2.4}$$



Now $z_k^{l+1} = \sum_{i=1}^{p} w_{ki}^{l+1} \cdot a_i^l + b_k^{l+1}$ and so $\partial z_k^{l+1}/\partial a_j^l = w_{kj}^{l+1}$. Therefore,

$$\frac{\partial C}{\partial a_j^l} = \sum_{k=1}^{q} \delta_k^{l+1} \cdot w_{kj}^{l+1}. \tag{2.5}$$

We can now write the error of the jth neuron in layer l as:

$$\delta_j^l = \sigma'(z_j^l) \cdot \sum_{k=1}^{q} \delta_k^{l+1} \cdot w_{kj}^{l+1}. \tag{2.6}$$

The above equation in matrix-form is precisely Equation (2.3).

Step III. Evaluating $\partial C/\partial b_j^l$

This is straightforward now.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} = \delta_j^l. \tag{2.7}$$

Step IV. Evaluating $\partial C/\partial w_{jk}^l$

This is also a straightforward calculation.

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \cdot \frac{\partial}{\partial w_{jk}^l} \left(\sum_{i=1}^{p} w_{ji}^l a_i^{l-1} + b_j^l \right) = \delta_j^l \cdot a_k^{l-1}. \tag{2.8}$$

We may write the backpropagation equations as:

$$\boxed{\begin{aligned}
\delta^L &= \nabla_{a^L} C \odot \sigma'(z^L) \\
\delta^l &= ((w^{l+1})^\intercal \delta^{l+1}) \odot \sigma'(z^l) \\
\frac{\partial C}{\partial b^l_j} &= \delta^l_j := \frac{\partial C}{\partial z^l_j} \\
\frac{\partial C}{\partial w^l_{jk}} &= a^{l-1}_k \delta^l_j
\end{aligned}} \tag{2.9}$$

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 $x_1, \dots, x_m$.

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,

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

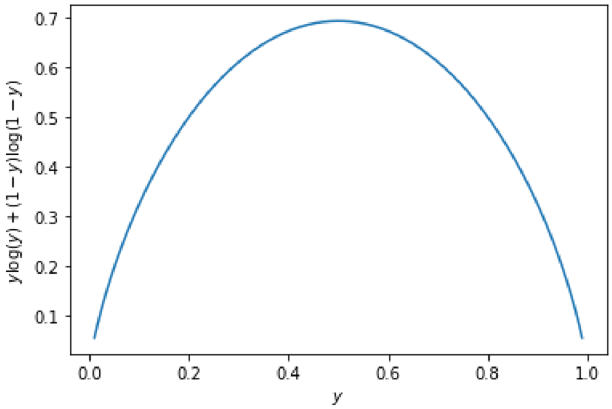


Figure 3.1: The Binary Entropy Function

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:

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

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:

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

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. ■

3.3 Regularization

Regularization is a set of methods used to prevent overfitting and improve generalization performance. The typical way to do this is to tack on a "regularization term" in the loss function. The most commonly used of these are the L_1-regularization and the L_2-regularization. These are defined in terms of the L_1 and L_2-norms respectively. For a real $p \geq 1$, the L_p-norm of a vector $w \in \mathrm{R}^d$ is defined as

$$\|w\| = (|w_1|^p + \cdots + |w_d|^p)^{1/p}. \tag{3.8}$$

Let us assume that $\mathscr{L}(w)$ is the original loss function. Then the L_2-regularized loss functions looks like this:

$$\mathscr{L}(w) + \frac{\alpha}{2} \cdot \|w\|_2^2 ,$$

where $\alpha > 0$ is real number that specifies the "strength" of the regularization.

3.3.1 L_2-regularization Analysis

Let $w^\star$ be an optimal solution to $\mathscr{L}(w)$, the unregularized loss function. Given a vector $w \in \mathrm{R}^d$, write w as $w^\star + \epsilon$, where ϵ is viewed as the "error." Use a second-order Taylor series expansion to write down the loss $\mathscr{L}$ at the point $w^\star + \epsilon$ as:

$$\mathscr{L}(w^\star + \epsilon) = \mathscr{L}(w^\star) + \epsilon^T \cdot \nabla\mathscr{L}(w^\star) + \frac{1}{2}\epsilon^T \cdot H_{\mathscr{L}} \cdot \epsilon, \tag{3.9}$$

where $H_{\mathscr{L}}$ represents the Hessian of $\mathscr{L}$ evaluated at $w^\star$. Recall that the Hessian, in this case, is a $d \times d$ matrix whose (i, j)th term is given by:

$$\left.\frac{\partial^2 \mathscr{L}(w)}{\partial w_i \partial w_j}\right|_{w^\star} .$$

We assume that the loss function is continuous at $w^\star$ so that the order of partial derivatives does not matter. In this case, the Hessian is a symmetric $d \times d$ matrix. In particular, note that this implies that all eigenvalues of the Hessian matrix are real.

Since $\nabla\mathscr{L}(w^\star) = 0$, we may rewrite Equation 3.9 as:

$$\mathscr{L}(w) = \mathscr{L}(w^\star) + \frac{1}{2}(w - w^\star)^T \cdot H_{\mathscr{L}} \cdot (w - w^\star).$$

Taking gradients wrt w, we obtain that:

$$\nabla\mathscr{L}(w) = 0 + H_{\mathscr{L}} \cdot (w - w^\star). \tag{3.10}$$

Consider the L_2-regularized loss function $\tilde{\mathscr{L}}(w) = \mathscr{L}(w) + \frac{\alpha}{2} \cdot \|w\|^2$. Let $\tilde{w}$ be an optimum solution to this regularized loss function. Then

$$0 = \nabla\tilde{\mathscr{L}}(\tilde{w}) = \nabla\mathscr{L}(\tilde{w}) + \alpha\tilde{w}. \tag{3.11}$$

Using the expression from Equation 3.10 in the above equation, we get:

$$\begin{aligned} H_{\mathscr{L}} \cdot (\tilde{w} - w^\star) + \alpha\tilde{w} &= 0 \\ (H_{\mathscr{L}} + \alpha \cdot I)\tilde{w} &= H_{\mathscr{L}} \cdot w^\star \\ \therefore\ \tilde{w} &= (H_{\mathscr{L}} + \alpha \cdot I)^{-1} \cdot H_{\mathscr{L}} \cdot w^\star. \end{aligned} \tag{3.12}$$

Here we are assuming that the symmetric matrix $H_{\mathscr{L}} + \alpha \cdot I$ is invertible. In fact, we are going make a stronger assumption about $H_{\mathscr{L}}$ that it is positive definite. Recall that this means that the eigenvalues of $H_{\mathscr{L}}$ are all strictly positive.

Any symmetric matrix A of order d can be written as $A = Q\Lambda Q^T$, where the columns of Q are the eigenvectors $q_1, \ldots, q_d$ of A and $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_d)$, where $Aq_i = \lambda_i q_i$. Moreover, the eigenvectors $q_1, \ldots, q_d$ form an orthonormal basis of R^d. In particular, this means that $Q^T Q = I$ so that $Q^{-1} = Q^T$. Substituting this **spectral decomposition** form for $H_{\mathscr{L}}$ in the last equation for $\tilde{w}$, we get:

$$\begin{aligned}
\tilde{w} &= (H_{\mathscr{L}} + \alpha I)^{-1} \cdot H_{\mathscr{L}} \cdot w^\star \\
&= (Q\Lambda Q^T + \alpha Q I Q^T)^{-1} \cdot Q\Lambda Q^T \cdot w^\star \\
&= (Q(\Lambda + \alpha I)Q^T)^{-1} \cdot Q\Lambda Q^T \cdot w^\star \\
&= Q(\Lambda + \alpha I)^{-1} Q^T \cdot Q\Lambda Q^T \cdot w^\star \\
&= Q(\Lambda + \alpha I)^{-1} \Lambda Q^T \cdot w^\star \\
&= Q \,\text{diag}\left(\frac{\lambda_1}{\lambda_1 + \alpha}, \ldots, \frac{\lambda_d}{\lambda_d + \alpha}\right) Q^T \cdot w^\star. \qquad (3.13)
\end{aligned}$$

This can be interpreted as follows. To obtain an optimal solution $\tilde{w}$ to the L_2-regularized problem, one proceeds as follows. One takes an optimal solution $w^\star$ to the unregularized loss function, applies an orthogonal transformation Q^T to it which has the effect of rotating it. Each component of that rotated vector is then scaled by the matrix $\text{diag}(\lambda_1/(\lambda_1 + \alpha), \ldots, \lambda_d/(\lambda_d + \alpha))$. But the scaling for component i equals $\lambda_i/(\lambda_i + \alpha)$ which depends on the eigenvalue λ_i. If $\lambda_i \gg \alpha$, then the fraction $\lambda_i/(\lambda_i + \alpha) \approx 1$ and there is no appreciable change in that component. But if $\lambda_i \ll \alpha$, then $\lambda_i/(\lambda_i + \alpha) \approx 0$ and that component effectively zeros out. Finally, a reverse orthogonal transform is applied to the scaled vector. Had the diagonal entries of the scaling matrix above been all the same, the effect would be to simply multiply $w^\star$ by this scaling factor. But different eigenvalues imply a different scaling for each component. In the end, the eigenvalues of $H_{\mathscr{L}}$ determine which components of the weight vector w are important.

3.3.2 Other Methods for Regularizing Neural Nets

Other than constrain the loss function, there are several other practical techniques to regulaize neural networks. These are:

1. Early stopping
2. Dataset augmentation
3. Noise injection
4. Dropout.

Early stopping

The ideas here is to continuously monitor the loss function and stop the training when the validation loss does not meet some previously defined criterion. An example of such a criterion is the *error change criterion*: pause training when the validation error does not drop over a window of, say, 10 epochs; continue training for a fixed number of epochs when this criterion is reached before finally stopping.

Another idea is the *weight change criterion*: compare the weights at epochs, say, $t-10$ and t and test whether

$$\max_i \left\| w_i^{(t-10)} - w_i^{(t)} \right\| \leq \rho,$$

where ρ is some predefined threshold. Instead of using an absolute value, one could use a threshold based on the percentage of the weight w_i^t.

Dataset Augmentation

This idea is particularly relevant when using neural networks to detect images. Certain transformations of an image do not change the label of the image. These include distorting or blurring the image, rotating it, changing its color, injecting noise, mirroring it, and so on and so forth. Using these transformations, one gets to increase the size of the training dataset. This has the effect of preventing overfitting as the neural network is exposed to data that is not explicitly present in the original dataset.

Another method of data augmentation is that of *mix up*. This consists in first choosing a parameter $\lambda \in [0,1]$, and then given a pair of data points $(x^{(i)}, y^{(i)})$ and $(x^{(j)}, y^{(j)})$, creating a new data point $(\tilde{x}, \tilde{y})$, which is the convex combination of these data points:

$$\tilde{x} = \lambda x^{(i)} + (1-\lambda)x^{(j)} \tag{3.14}$$

$$\tilde{y} = \lambda y^{(i)} + (1-\lambda)y^{(j)}. \tag{3.15}$$

Noise Injection

Here the idea is to add noise to either the input data or the labels or the gradients during training. Let's look at the case where one adds Gaussian noise to the input data during training when the loss function is the expected squared-loss.

As usual, let the data points be $(x, y) \in \mathrm{R}^d \times \mathrm{R}$ and let us assume that the joint density of the input-output pairs is given by the function $f_{X,Y} \colon \mathrm{R}^d \times \mathrm{R} \longrightarrow [0,1]$. Let us add Gaussian noise $\epsilon \sim N_d(0_d, \Sigma_d)$, where 0_d is the zero-vector from R^d and $\Sigma_d = \operatorname{diag}(\sigma^2, \ldots, \sigma^2)$. Recall that if a random vector has a multivariate normal distribution, then two components of that vector are independent iff they are uncorrelated. This is *not* true for random variables X, Y, in general, even if both are normally distributed. In this specific case, this implies that for all $1 \leq i, j \leq d$:

$$\mathrm{E}\left[\epsilon_i \epsilon_j\right] = \begin{cases} 0 & \text{if } i \neq j \\ \sigma^2 & \text{if } i = j. \end{cases} \tag{3.16}$$

The expected squared-loss is:

$$\mathscr{L}(w) = \mathrm{E}_{(x,y)\sim f_{X,Y}}\left[(w^T x - y)^2\right] = \iint (w^T x - y)^2 \cdot f_{X,Y}(x,y)\mathrm{d}y\mathrm{d}x. \tag{3.17}$$

The expected squared-loss error in the presence of noise is given by:

$$\begin{aligned}\tilde{\mathscr{L}}(w) &= \mathrm{E}_{\epsilon\sim N_d,(x,y)\sim f_{X,Y}}\left[(w^T(x+\epsilon)-y)^2\right] \\ &= \iint \mathrm{E}_{\epsilon\sim N_d}\left[(w^T(x+\epsilon)-y)^2 \mid (x,y)\right]\cdot f_{X,Y}(x,y)\mathrm{d}y\mathrm{d}x. \end{aligned} \tag{3.18}$$

Simplifying the expression for the loss in Equation (3.18), we get that:

$$\begin{aligned}\tilde{\mathscr{L}}(w) &= \iint \mathrm{E}_{\epsilon\sim N_d}\left[(w^Tx-y)^2+2(w^Tx-y)(w^T\epsilon)+(w^T\epsilon)^2 \mid (x,y)\right] f_{X,Y}(x,y)\mathrm{d}y\mathrm{d}x \\ &= \iint \left[(w^Tx-y)^2+2(w^Tx-y)\mathrm{E}\left[(w^T\epsilon)\mid(x,y)\right]+\mathrm{E}\left[(w^T\epsilon)^2\mid(x,y)\right]\right] f_{X,Y}(x,y)\mathrm{d}y\mathrm{d}x \end{aligned} \tag{3.19}$$

The expectations inside the integral sign in the above equation are all wrt the distribution of ϵ. Now,

$$\begin{aligned}\mathrm{E}_{\epsilon\sim N_d}\left[(w^T\epsilon)\mid(x,y)\right] &= \mathrm{E}_{\epsilon\sim N_d}\left[(w^T\epsilon)\right] \\ &= \mathrm{E}_{\epsilon\sim N_d}\left[w_1\epsilon_1+\cdots+w_d\epsilon_d\right] \\ &= 0. \end{aligned} \tag{3.20}$$

Moreover,

$$\begin{aligned}\mathrm{E}_{\epsilon\sim N_d}\left[(w^T\epsilon)^2\mid(x,y)\right] &= \mathrm{E}_{\epsilon\sim N_d}\left[(w^T\epsilon)^2\right] \\ &= \mathrm{E}_{\epsilon\sim N_d}\left[\sum_{i=1}^d w_i^2\epsilon_i^2+2\cdot\sum_{i<j}w_iw_j\epsilon_i\epsilon_j\right] \\ &= \sum_{i=1}^d w_i^2\sigma^2 \\ &= \sigma^2\sum_{i=1}^d w_i^2. \end{aligned} \tag{3.21}$$

From Equations 3.19, 3.20 and 3.21, we obtain that:

$$\tilde{\mathscr{L}}(w) = \mathscr{L}(w)+\sigma^2\sum_{i=1}^d w_i^2, \tag{3.22}$$

which is the expression for the loss function regularized with L_2-loss.

Dropout

The idea here is to randomly zero out a certain fraction p of nodes from each hidden layer for each training sample and each iteration during the training phase. The nodes that are “zeroed out” have their activations set to 0. In the test phase, all activations are used but all of them are reduced by a factor of p to account for the missing fraction during training.

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.

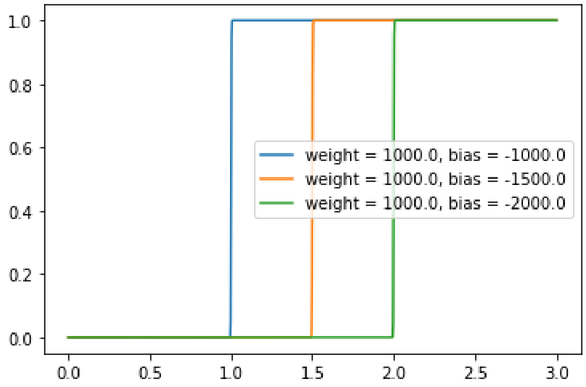


Figure 4.1: Output of a single sigmoid neuron for different values of weight and bias

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 $\mathrm{R}^m \to \mathrm{R}^n$. The output of such a network is shown in Figure 4.3.

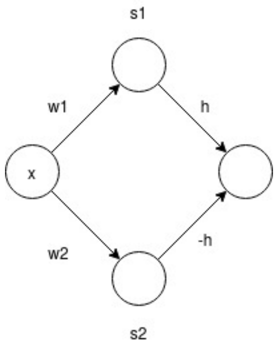


Figure 4.2: Network of sigmoid neurons to output a rectangle function.

4.2 Functions of One Variable

The network described in the last section acts as an indicator 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 $[s_1, s_2]$, $[s_2, s_3]$, $[s_3, s_4]$

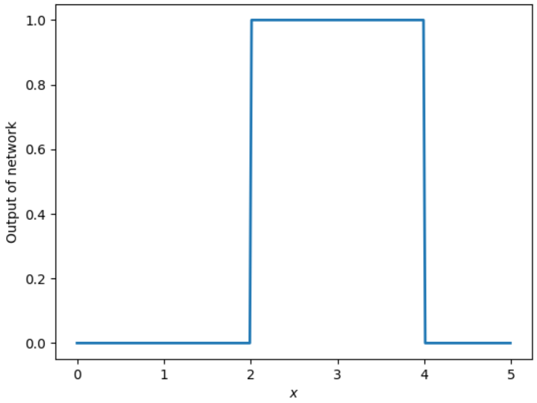


Figure 4.3: Output of the network. $w_1 = w_2 = 10000.0$ and $h = 10000.0$

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.

We created a small program that approximates the function provided in Nielsen's book: $f(x) = 0.2 + 0.4x^2 + 0.3x\sin(15x) + 0.05\cos(50x)$. We show some of the results in Figures 4.5.

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 $f\colon \mathrm{R}^2 \to \mathrm{R}^1$.

The network used to approximate a step function in R^2 is shown in Figure 4.6. This network has two sub-networks: one for creating a step function along the x-axis and another for the y-axis. The construction of these sub-networks is exactly the same as in the case of one-variable functions with the difference being that the output neuron now has to output a 1 iff $(x, y) \in (s_x^1, s_x^2) \times (s_y^1, s_y^2)$. This is achieved by adjusting the bias of the output neuron to $-3h/2$, where h is some large number.

Let us look at this network a little more closely. The top two neurons of the first hidden layer labelled s_x^1 and s_x^2 to indicate the fact that they help create the indicator function for the interval (s_x^1, s_x^2) on the x-axis. Let their input weight and bias be w_1, b_1 and w_2, b_2, respectively. Similarly,

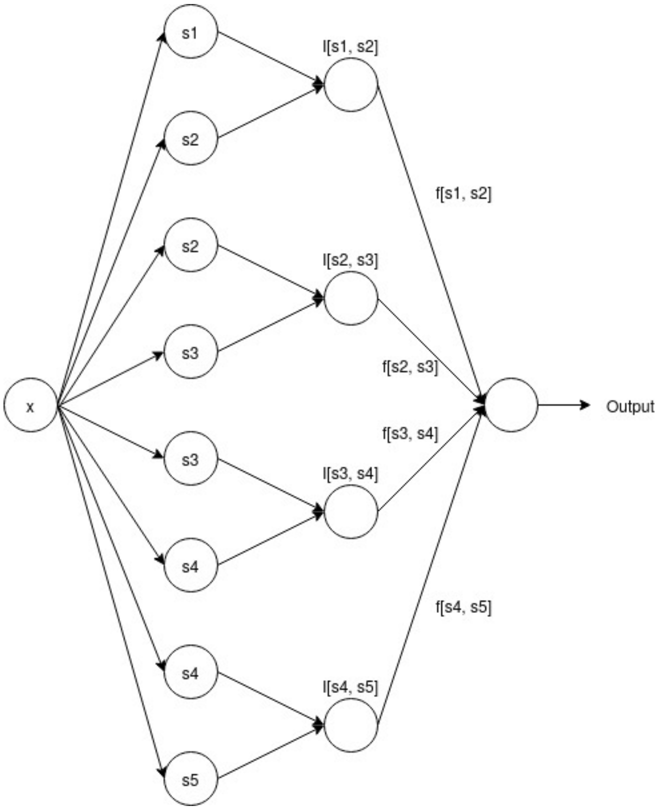


Figure 4.4: Network that approximates a function on the range $[s_1, s_5]$

the bottom two neurons of the first hidden layer and labelled s_y^1 and s_y^2. Let their input weight and bias be w_3, b_3 and w_4, b_4, respectively. Now if the input to the network is (x, y), the weighted input to the output neuron is given by:

$$h \cdot \sigma(w_1 x + b_1) - h \cdot \sigma(w_2 x + b_2) + h \cdot \sigma(w_3 y + b_3) - h \cdot \sigma(w_4 y + b_4).$$

The crucial observation here is that when $x \in (s_x^1, s_x^2)$ and $y \in (s_y^1, s_y^2)$ then this weighted input is $\approx 2h$; otherwise, this is $< h$. Hence the output in the first case is $\sigma(2h - 3h/2) = \sigma(h/2) \approx 1$, whereas in the second case is $\sigma(h - 3h/2) = \sigma(-h/2) \approx 0$. This holds whenever h is some very large number. The bias can be any negative number whose absolute value is between h and $2h$. We just chose $-3h/2$.

Also note that we can extend this naturally for functions of three or more variables. For three variables, we would have another sub-network that creates an indicator for an interval (s_z^1, s_z^2) on the z-axis. This sub-network would then be wired to the output neuron whose bias would have to be modified to $-5h/2$. Again the bias can be any number whose absolute value is between $2h$ and $3h$. For an m-variable function, we will have m sub-networks, each creating an indicator function for an interval along the appropriate axis, connected to the output neuron with bias $-(h + h - 1)/2 = -(2h - 1)/2$. These can then be connected with other sub-networks for an extended range of intervals to approximate a function on this range of intervals. A plot of the tower function created by this network for the rectangular region $(1.0, 2.0) \times (-2.0, 0.0)$ is shown

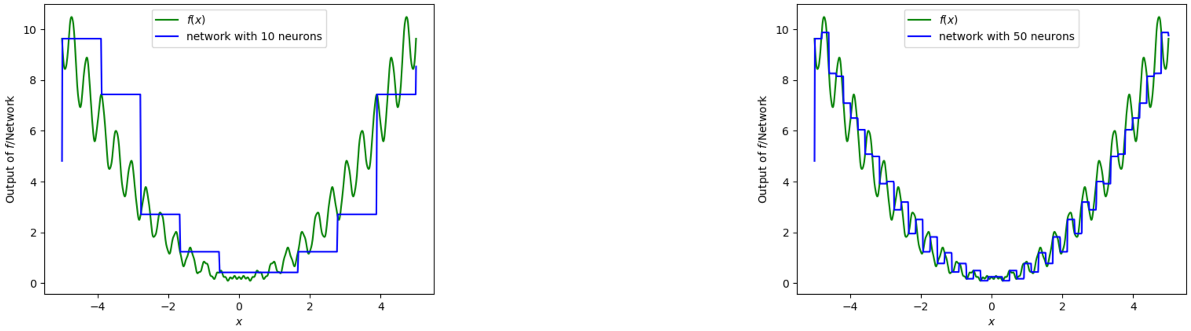


Figure 4.5: Approximating with 10 neurons and 50 neurons.

in Figure 4.7.

Finally, we consider the case of functions f from R^m to R^n. Any such function can be decomposed into n functions $f_1(x_1,\ldots,x_m),\ldots,f_n(x_1,\ldots,x_m)$ from $\mathrm{R}^m \to \mathrm{R}^1$. Hence this case can be handled by combining n networks one for each of the n functions.

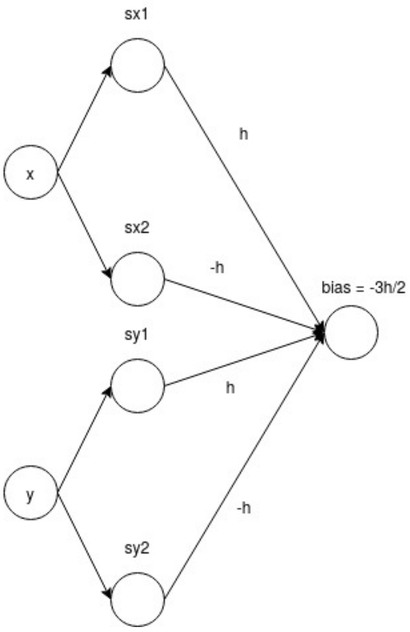


Figure 4.6: Network that approximates a tower function in two variables

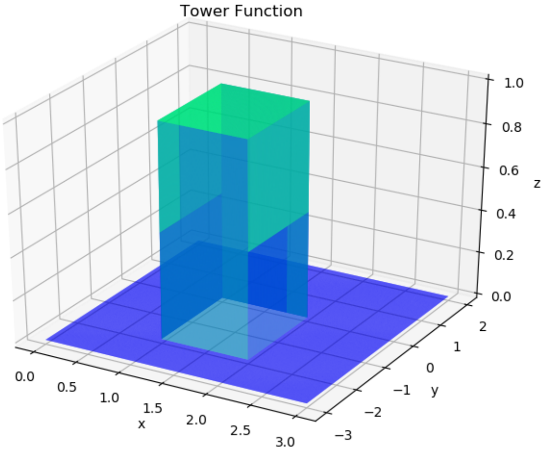


Figure 4.7: Plot of the tower function created by the network in Figure 4.6.

Chapter 5

RNNs and LSTMs

5.1 Vanishing and Exploding Gradients

Training neural networks using backpropagation requires us to compute the partial derivatives of the loss function wrt each weight and bias of the network. The weights and biases are then revised using these partial derivatives using the stochastic gradient descent formulation:

$$w^{\text{new}} = w^{\text{old}} - \eta \frac{\partial L}{\partial w}.$$

If the network is deep enough, then in certain cases it possible for the term $\partial L / \partial w$ to be very small or very large. If this term is very small, then the weight is not updated; if it is very large, there is a large update to the weight, which keeps on repeating at each updation step, with the result that the weights do not converge.

To illustrate this problem, consider a simple network with L layers, with two neurons in each of the first $L - 1$ layers and a single neuron in the final layer L. This network is a map from $\mathrm{R}^2 \rightarrow \mathrm{R}^1$. Let $W^{(i)}$ be the weight matrix associated with layer i, where $W^{(1)}, \ldots, W^{(L-1)} \in \mathrm{R}^{2\times 2}$ and $W^{(L)} \in \mathrm{R}^{1\times 2}$. Ignore the biases and assume that there are no activation functions. Consider a single input-output pair $((x_1, x_2)^T, y)$. Denote the input to layer i as $\boldsymbol{x}^{(i-1)}$, so that input to the very first layer is $\boldsymbol{x}^{(0)} = (x_1, x_2)^T$. Then the output of layer i, ignoring the bias and the activation, is $W^{(i)}\boldsymbol{x}^{(i-1)}$. The output of the first layer is $W^{(1)}\boldsymbol{x}^{(0)}$, that of the second is $W^{(2)}W^{(1)}\boldsymbol{x}^{(0)}$ and so on. The final output $\hat{y}$ of the network is easily seen to be:

$$\hat{y} = W^{(L)} \cdot W^{(L-1)} \cdots W^{(1)} \cdot \boldsymbol{x}^{(0)}.$$

To write this final output more explicitly in terms of the individual weights, we need a little more notation. Let $W^{(i)}$, for $1 \leq i \leq L - 1$, be:

$$\begin{pmatrix} w_{11}^{(i)} & 0 \\ 0 & w_{22}^{(i)} \end{pmatrix}.$$

Let $W^{(L)} = (w_{11}^{(L)}, w_{12}^{(L)})$. Then

$$\hat{y} = (w_{11}^{(L)}, w_{12}^{(L)}) \cdot \begin{pmatrix} w_{11}^{(L-1)} w_{11}^{(L-2)} \cdots w_{11}^{(2)} w_{11}^{(1)} x_1 \\ w_{22}^{(L-1)} w_{22}^{(L-2)} \cdots w_{11}^{(2)} w_{22}^{(1)} x_2 \end{pmatrix}.$$

This ultimately yields:

$$\hat{y} = w_{11}^{(L)} w_{11}^{(L-1)} w_{11}^{(L-2)} \cdots w_{11}^{(2)} w_{11}^{(1)} x_1 + w_{12}^{(L)} w_{22}^{(L-1)} w_{22}^{(L-2)} \cdots w_{11}^{(2)} w_{22}^{(1)} x_2.$$

Assume that the loss function is the standard squared loss: $L(y, \hat{y}) = 0.5 \cdot (y - \hat{y})^2$. Consider the derivative of L wrt one of the weights in the very first layer.

$$\begin{aligned} \frac{\partial L}{\partial w_{11}^{(1)}} &= \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{11}^{(1)}} \\ &= (\hat{y} - y) \cdot w_{11}^{(L)} w_{11}^{(L-1)} \cdots w_{11}^{(2)} x_1. \end{aligned}$$

Notice that the partial derivative is a product that tacks on weights, one from each layer, starting with the second layer. Now if L is large and these weights were all slightly less than 1, then resulting product would be very small; if these weights are slightly larger than 1, then the product would be a large quantity. The first case is the *vanishing gradient* problem; the second is the *exploding gradient* problem. Note that is does not matter which weight we use to differentiate the loss function. The partial derivative product term picks up weights form each layer (except the one wrt which we are differentiating).

Exercise 5.1. Can we avoid the vanishing gradient problem by using a different activation function than the sigmoid function?

Solution. If we use an activation function f whose derivative is higher at $f'(0)$ is higher, but less than 1, we can reduce the severity of the vanishing gradient problem, but not eliminate it altogether. If we were choose f such that $f'(0) > 1$, then we might end up with the exploding gradient problem.

One such candidiate function is the hyperbolic tangent function. This is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

The derivative of this function is $\tanh'(x) = 1 - \tanh(x)^2$. The graph of the function and its derivative is shown in Figure 5.1.

■

5.2 LSTMs

These notes are based on [1, 2]. LSTMs were developed in order to circumvent the vanishing gradient problem that plagues multi-layered RNNs. LSTMs are equipped with a long-term memory and a short-term working memory.

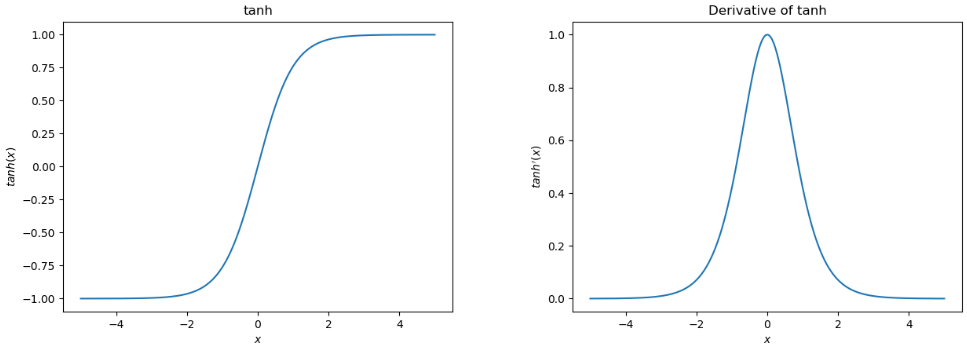


Figure 5.1: The Tanh function and its derivative

Let $x_t \in \mathrm{R}^p$ denote the input at time t; let $\mathrm{ltm}_t \in \mathrm{R}^d$ and $\mathrm{wm}_t \in \mathrm{R}^d$ denote, respectively, the long-term memory and the working memory available to the LSTM cell at time t.

Updating the long-term memory. In order to update the long-term memory at step t, the LSTM first figures out what to remember from the long-term memory of the last step $t-1$.

$$\mathrm{rem}_t = \sigma\left(W_r \cdot x_t + U_r \cdot \mathrm{wm}_{t-1} + b_r\right). \tag{5.1}$$

This is accomplished using a one-layer neural network with a sigmoid activation function that estimates the weight matrices $W_r \in \mathrm{R}^{p\times d}$, $U_r \in \mathrm{R}^{d\times d}$ and the bias vector $b_r \in \mathrm{R}^d$. Since the activation function is sigmoid, the components of rem_t are between 0 and 1. If a component is closer to 1, we would want to remember it; if it is close to 0, then we want to forget it.

It next calculates a "candidate" vector to add to its long-term memory. This is done using a single-layer neural network with a tanh activation function. Denote this candidate by ltm'_t.

$$\mathrm{ltm}'_t = \tanh\left(W_l \cdot x_t + U_l \cdot \mathrm{wm}_{t-1} + b_l\right). \tag{5.2}$$

As usual, $W_l \in \mathrm{R}^{p\times d}$, $U_l \in \mathrm{R}^{d\times d}$ and $b_l \in \mathrm{R}^d$.

Not all parts of this candidate vector may be worth remembering. As such, a save_t vector is created using another single-layer neural network with a sigmoid activation function.

$$\mathrm{save}_t = \sigma\left(W_s \cdot x_t + U_s \cdot \mathrm{wm}_{t-1} + b_s\right). \tag{5.3}$$

The dimensions of the weight matrices W_s, U_s and the bias vector b_s are such that $\mathrm{save}_t \in \mathrm{R}^d$. Now the long-term component of the cell is computed using:

$$\mathrm{ltm}_t = \mathrm{rem}_t \odot \mathrm{ltm}_{t-1} + \mathrm{save}_t \odot \mathrm{ltm}'_t, \tag{5.4}$$

where $\odot$ represents component-wise multiplication of the d-dimensional vectors.

Updating the working memory. To do this, the LSTM first calculates what parts of the long-term memory it currently wants to focus on. It uses another single-layer neural network with a sigmoid activation to calculate $\mathrm{focus}_t \in \mathrm{R}^d$.

$$\mathrm{focus}_t = \sigma\left(W_f \cdot x_t + U_f \cdot \mathrm{wm}_{t-1} + b_f\right). \tag{5.5}$$

It then updates its working memory using:

$$\mathrm{wm}_t = \mathrm{focus}_t \odot \tanh(\mathrm{ltm}_t). \tag{5.6}$$

$$\begin{aligned}
\mathrm{rem}_t &= \sigma\left(W_r \cdot x_t + U_r \cdot \mathrm{wm}_{t-1} + b_r\right)\\
\mathrm{save}_t &= \sigma\left(W_s \cdot x_t + U_s \cdot \mathrm{wm}_{t-1} + b_s\right)\\
\mathrm{focus}_t &= \sigma\left(W_f \cdot x_t + U_f \cdot \mathrm{wm}_{t-1} + b_f\right)\\
\mathrm{ltm}'_t &= \tanh\left(W_l \cdot x_t + U_l \cdot \mathrm{wm}_{t-1} + b_l\right)\\
\mathrm{ltm}_t &= \mathrm{rem}_t \odot \mathrm{ltm}_{t-1} + \mathrm{save}_t \odot \mathrm{ltm}'_t\\
\mathrm{wm}_t &= \mathrm{focus}_t \odot \tanh(\mathrm{ltm}_t).
\end{aligned}$$

Figure 5.2: All the LSTM equations at once.

Bibliography

[1] Edwin Chen. *Exploring LSTMs*. Available at: `http://blog.echen.me/2017/05/30/exploring-lstms/`.

[2] Christopher Olah. *Understanding LSTM Networks*. Available at: `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.