Neural Networks

Matt R

March 3, 2022

Contents

Part I

Neural Networks and Deep Learning

1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have we have training examples $x \in \mathbb{R}^{m \times n}$ with binary labels $y \in \{0, 1\}^{1 \times n}$. We desire to train a model which yields an output a which represents

$$a = \mathbb{P}(y = 1|x).$$

To this end, let $\sigma : \mathbb{R} \to (0, 1)$ denote the sigmoid function, i.e.,

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

and let $w \in \mathbb{R}^m$, $b \in \mathbb{R}$, and let

$$a = \sigma(w^T x + b).$$

To analyze the accuracy of model, we need a way to compare y and a, and ideally this functional comparison can be optimized with respect to (w, b) in such a way to minimize the error. To this end, we note that

$$\mathbb{P}(y|x) = a^y (1 - a)^{1-y},$$

or rather

$$\mathbb{P}(y = 1|x) = a, \qquad \mathbb{P}(y = 0|x) = 1 - a,$$

so $\mathbb{P}(y|x)$ represents the corrected probability. Now since we want

$$a \approx 1 \quad \text{when } y = 1,$$

and

$$a \approx 0 \quad \text{when } y = 0,$$

and $0 \le a \le 1$, any error using differences won't be refined enough to analyze when tuning the model. Moreover, since introducing the sigmoid function, our usual mean-squared-error function won't be convex. This leads us to apply the log function, which when restricted to $(0, 1)$ is a bijective mapping of $(0, 1) \to (-\infty, 0)$. This leads us to define our log-loss function

$$\begin{aligned}
\mathbb{L}(a, y) &= -\log(\mathbb{P}(y|x)) \\
&= -\log\left(a^y (1 - a)^{1-y}\right) \\
&= -\left[y \log(a) + (1 - y) \log(1 - a)\right],
\end{aligned}$$

and finally, since we wish to analyze how our model performs on the entire training set, we need to average our log-loss functions to obtain our cost function $\mathbb{J}$ defined by

$$\begin{aligned}
\mathbb{J}(w, b) &= \frac{1}{n} \sum_{j=1}^{n} \mathbb{L}(a_j, y_j) \\
&= -\frac{1}{n} \sum_{j=1}^{n} \left[y_j \log(a_j) + (1 - y_j) \log(1 - a_j) \right] \\
&= -\frac{1}{n} \sum_{j=1}^{n} \left[y_j \log(\sigma(w^T x_j + b)) + (1 - y_j) \log(1 - \sigma(w^T x_j + b)) \right].
\end{aligned}$$

1.1 The Gradient

To compute the gradient of our cost function $\mathbb{J}$, we first write $\mathbb{J}$ as a sum of compositions as follows: We have the log-loss function considered as a map $\mathbb{L} : (0, 1) \times \mathbb{R} \to \mathbb{R}$,

$$\mathbb{L}(a, y) = - \left[y \log(a) + (1 - y) \log(1 - a) \right],$$

we have the sigmoid function $\sigma : \mathbb{R} \to (0, 1)$ with $\sigma(z) = a$ and $\sigma'(z) = a(1 - a)$, and we have the collection of affine-functionals $\phi_x : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ given by

$$\phi_x(w, b) = w^T x + b,$$

for which we fix an arbitrary $x \in \mathbb{R}^m$ and write $\phi = \phi_x$, and set $z = \phi(w, b)$. Finally, we introduce the auxiliary function $\mathcal{L} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ given by

$$\mathcal{L}(w, b) = \mathbb{L}(\sigma(\phi(w, b)), y).$$

Then by the chain rule, we have that

$$\begin{aligned}
d\mathcal{L} &= d_a \mathbb{L}(a, y) \circ d\sigma(z) \circ d_w \phi(w, b) \\
&= \left[-\frac{y}{a} + \frac{1 - y}{1 - a} \right] \cdot a(1 - a) \cdot \begin{bmatrix} x^T & 1 \end{bmatrix} \\
&= \left[-y(1 - a) + a(1 - y) \right] \cdot \begin{bmatrix} x^T & 1 \end{bmatrix} \\
&= (a - y) \begin{bmatrix} x^T & 1 \end{bmatrix}
\end{aligned}$$

Composition turns into matrix multiplication in the tangent space.

4

Moreover, since in Euclidean space, we have that $\nabla f = (df)^T$, and hence that

$$\nabla \mathcal{L}(w, b) = (a - y) \begin{bmatrix} x \\ 1 \end{bmatrix},$$

or rather

$$\partial_w \mathbb{L}(a, y) = (a - y)x, \qquad \partial_b \mathbb{L}(a, y) = a - y.$$

Finally, since our cost function $\mathbb{J}$ is the sum-log-loss, we have by linearity that

$$\partial_w \mathbb{J}(w, b) = \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j)x_j$$

$$= \frac{1}{n}((a - y) \cdot x^T)^T$$

$$= \frac{1}{n}x \cdot (a - y)^T$$

and

$$\partial_b \mathbb{J}(w, b) = \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j).$$

1.1.1 Vectorization in Python

Here we include the general code to train a model using logistic regression without regularization and without tuning on a cross-validation set.

```python
import copy

import numpy as np

def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
    """

    sigma = (1 / (1 + np.exp(-z)))
    return sigma

```

```python
19  def cost_function(x, y, w, b):
20      """
21      Parameters
22      ----------
23      x : array_like
24          x.shape = (m, n) with m-features and n-examples
25      y : array_like
26          y.shape = (1, n)
27      w : array_like
28          w.shape = (m, 1)
29      b : float
30
31      Returns
32      -------
33      J : float
34          The value of the cost function evaluated at (w, b)
35      dw : array_like
36          dw.shape = w.shape = (m, 1)
37          The gradient of J with respect to w
38      db : float
39          The partial derivative of J with respect to b
40      """
41
42      # Auxiliary assignments
43      m, n = x.shape
44      z = w.T @ x + b
45      assert z.size == n
46      a = sigmoid(z).reshape(1, n)
47      dz = a - y
48
49      # Compute cost J
50      J = (-1 / n) * (np.log(a) @ y.T + np.log(1 - a) @ (1 - y).T)
51
52      # Compute dw and db
53      dw = (x @ dz.T) / m
54      assert dw.shape == w.shape
55      db = np.sum(dz) / m
56
57      return J, dw, db
58
59  def grad_descent(x, y, w, b, alpha=0.001, num_iters=2000, print_cost=False):
60      """
61      Parameters
62      ----------
63      x, y, w, b : See cost_function above for specifics.
64          w and b are chosen to initialize the descent (likely all components 0)
65      alpha : float
```

```
66          The learning rate of gradient descent
67      num_iters : int
68          The number of times we wish to perform gradient descent
69
70      Returns
71      -------
72      costs : List[float]
73          For each iteration we record the cost-values associated to (w, b)
74      params : Dict[w : array_like, b : float]
75          w : array_like
76              Optimized weight parameter w after iterating through grad descent
77          b : float
78              Optimized bias parameter b after iterating through grad descent
79      grads : Dict[dw : array_like, db : float]
80          dw : array_like
81              The optimized gradient with repsect to w
82          db : float
83              The optimized derivative with respect to b
84      """

85
86      costs = []
87      w = copy.deepcopy(w)
88      b = copy.deepcopy(b)
89      for i in range(num_iters):
90          J, dw, db = cost_function(x, y, w, b)
91          w = w - alpha * dw
92          b = b - alpha * db
93
94          if i % 100 == 0:
95              costs.append(J)
96              if print_cost:
97                  idx = int(i / 100) - 1
98                  print(f'Cost after iteration {i}: {costs[idx]}')
99
100     params = {'w' : w, 'b' : b}
101     grads = {'dw' : dw, 'db' : db}
102
103     return costs, params, grads
104
105 def predict(w, b, x):
106     """
107     Parameters
108     ----------
109     w : array_like
110         w.shape = (m, 1)
111     b : float
112     x : array_like
```

```
113        x.shape = (m, n)
114
115    Returns
116    -------
117    y_predict : array_like
118        y_pred.shape = (1, n)
119        An array containing the prediction of our model applied to training
120        data x, i.e., y_pred = 1 or y_pred = 0.
121    """
122
123    m, n = x.shape
124    # Get probability array
125    a = sigmoid(w.T @ x + b)
126    # Get boolean array with False given by a < 0.5
127    pseudo_predict = ~(a < 0.5)
128    # Convert to binary to get predictions
129    y_predict = pseudo_predict.astype(int)
130
131    return y_predict
132
133 def model(x_train, y_train, x_test, y_test, alpha=0.001, num_iters=2000, accuracy=Tr
134    """
135    Parameters:
136    -----------
137    x_train, y_train, x_test, y_test : array_like
138        x_train.shape = (m, n_train)
139        y_train.shape = (1, n_train)
140        x_test.shape = (m, n_test)
141        y_test.shape = (1, n_test)
142    alpha : float
143        The learning rate for gradient descent
144    num_iters : int
145        The number of times we wish to perform gradient descent
146    accuracy : Boolean
147        Use True to print the accuracy of the model
148
149    Returns:
150    d : Dict
151        d['costs'] : array_like
152            The costs evaluated every 100 iterations
153        d['y_train_preds'] : array_like
154            Predicted values on the training set
155        d['y_test_preds'] : array_like
156            Predicted values on the test set
157        d['w'] : array_like
158            Optimized parameter w
159        d['b'] : float
```

```python
160              Optimized parameter b
161          d['learning_rate'] : float
162              The learning rate alpha
163          d['num_iters'] : int
164              The number of iterations with which gradient descent was performed
165
166      """
167
168      m = x_train.shape[0]
169      # initialize parameters
170      w = np.zeros((m, 1))
171      b = 0.0
172      # optimize parameters
173      costs, params, grads = grad_descent(x_train, y_train, w, b, alpha, num_iters)
174      w = params['w']
175      b = params['b']
176      # record predictions
177      y_train_preds = predict(w, b, x_train)
178      y_test_preds = predict(w, b, x_test)
179      # group results into dictionary for return
180      d = {'costs' : costs,
181          'y_train_preds' : y_train_preds,
182          'y_test_preds' : y_test_preds,
183          'w' : w,
184          'b' : b,
185          'learning_rate' : alpha,
186          'num_iters' : num_iters}
187
188      if accuracy:
189          train_acc = 100 - np.mean(np.abs(y_train_preds - y_train)) * 100
190          test_acc = 100 - np.mean(np.abs(y_test_preds - y_test)) * 100
191          print(f'Training_Accuracy:_{train_acc}%')
192          print(f'Test_Accuracy:_{test_acc}%')
193
194
195      return d
```

2   Neural Networks: A Single Hidden Layer

Suppose we wish to consider the binary classification problem given the training set (x, y) with $x \in \mathbb{R}^{s_0 \times n}$ and $y \in \{0, 1\}^{1 \times n}$. Usually with logistic regression we have the following type of structure:

$$[x^1, ..., x^{s_0}] \xrightarrow{\varphi} [z] \xrightarrow{g} [a] \xrightarrow{=} \hat{y},$$

where

$$z = \varphi(x) = w^T x + b,$$

is our affine-linear transformation, and

$$a = g(z) = \sigma(z)$$

is our sigmoid function. Such a structure will be called a *network*, and the $[a]$ is known as the *activation node*. Logistic regression can be too simplistic of a model for many situations, e.g., if the dataset isn't linearly separable (i.e., there doesn't exist some well-defined decision boundary built from a linear-surface), then logistic regression won't give a high-accuracy model. To modify this model to handle more complex situations, we introduce a new "hidden layer" of nodes with their own (possibly different) activation functions. That is, we consider a network of the following form:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

where

$$\varphi^{[1]} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_1}, \qquad \varphi^{[1]}(x) = W^{[1]}x + b^{[1]},$$

$$\varphi^{[2]} : \mathbb{R}^{s_1} \to \mathbb{R}, \qquad \varphi^{[2]}(x) = W^{[2]}x + b^{[2]},$$

and $W^{[1]} \in \mathbb{R}^{s_1 \times s_0}, W^{[2]} \in \mathbb{R}^{1 \times s_1}, b^{[1]} \in \mathbb{R}^{s_1}, b^{[2]} \in \mathbb{R}$, and $g^{[\ell]}$ is a *broadcasted* activator function (e.g., the sigmoid function $\sigma(z)$, or $\tanh(z)$, or $\text{ReLU}(z)$). Such a network is called a 2-layer neural network where x is the input layer (called layer-0), $a^{[1]}$ is a hidden layer (called layer-1), and $a^{[2]}$ is the output layer (called layer-2).

Definition 2.1. *Suppose* $g : \mathbb{R} \to \mathbb{R}$ *is any function. Then we say* $G : \mathbb{R}^m \to \mathbb{R}^m$ *is the* **broadcast** *of* g *from* $\mathbb{R}$ *to* $\mathbb{R}^m$ *if*

$$G(v) = G(v^i e_i)$$
$$= g(v^i)e_i,$$

where $v \in \mathbb{R}^m$ *and* $\{e_i : 1 \leq i \leq m\}$ *is the standard basis for* $\mathbb{R}^m$. *In practice, we will write* $g = G$ *for a broadcasted function, and let the context determine the meaning of* g.

<div style="border:1px solid black; display:inline-block">castingDifferential</div> **Lemma 2.2.** *Suppose* $g : \mathbb{R} \to \mathbb{R}$ *is any smooth function and* $G : \mathbb{R}^m \to \mathbb{R}^m$ *is the broadcasting of* g *from* $\mathbb{R}$ *to* $\mathbb{R}^m$. *Then the differential* $dG_z : T_z\mathbb{R}^m \to T_{G(z)}\mathbb{R}^m$ *is given by*

$$dG_z(v) = [g'(z^i)] \odot [v^i],$$

where $\odot$ *is the Hadamard product (also know as component-wise multiplication), and has matrix-representation in* $\mathbb{R}^{m \times m}$ *given by*

$$[dG_z]^i_j = \delta^i_j g'(z^i).$$

Proof: We calculate

$$dG_z(v) = \left.\frac{d}{dt}\right|_{t=0} G(z + tv)$$
$$= \left.\frac{d}{dt}\right|_{t=0} (g(z^i + tv^i))$$
$$= (g'(z^i)v^i)$$
$$= [g'(z^i)] \odot [v^i],$$

and letting $e_1, ... e_m$ denote the usual basis for $T_z\mathbb{R}^m$ (identified with $\mathbb{R}^m$), we see that

$$dG_z(e_j) = [g'(z^i)] \odot e_j$$
$$= g'(z^j)e_j,$$

from which conclude that dG_z is diagonal with (j, j)-th entry $g'(z^j)$ as desired. $\qquad\square$

Returning to our network, let us lay out all of these functions explicitly (in the Smooth Category) as to facilitate our later computations for our cost function and our gradients. To this end:

$$\varphi^{[1]} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_1}, \qquad\qquad d\varphi^{[1]} : T\mathbb{R}^{s_0} \to T\mathbb{R}^{s_1},$$
$$z^{[1]} = \varphi^{[1]}(x) = W^{[1]}x + b^{[1]}, \qquad\qquad d\varphi^{[1]}_x(v) = W^{[1]}v;$$

$$g^{[1]} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_1}, \qquad\qquad dg^{[1]} : T\mathbb{R}^{s_1} \to T\mathbb{R}^{s_1},$$

$$a^{[1]} = g^{[1]}(z^{[1]}), \qquad\qquad \frac{\partial a^{[1]\mu}}{\partial z^{[1]\nu}} = \delta^\mu_\nu g^{[1]\prime}(z^{[1]\mu});$$

$$\varphi^{[2]} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_2}, \qquad\qquad d\varphi^{[2]} : T\mathbb{R}^{s_1} \to T\mathbb{R}^{s_2},$$

$$z^{[2]} = \varphi^{[2]}(a^{[1]}) = W^{[2]}a^{[1]} + b^{[2]}, \qquad\qquad d\varphi^{[2]}_{a^{[2]}}(v) = W^{[2]}v;$$

$$g^{[2]} : \mathbb{R}^{s_2} \to \mathbb{R}^{s_2}, \qquad\qquad dg^{[2]} : T\mathbb{R}^{s_2} \to T\mathbb{R}^{s_2},$$

$$a^{[2]} = g^{[2]}(z^{[2]}), \qquad\qquad \frac{\partial a^{[2]\mu}}{\partial z^{[2]\nu}} = \delta^\mu_\nu g^{[2]\prime}(z^{[2]\mu}).$$

That is, given an input $x \in \mathbb{R}^{s_0}$, we get a predicted value $\hat{y} \in \mathbb{R}^{s_2}$ of the form

$$\hat{y} = g^{[2]} \circ \varphi^{[2]} \circ g^{[1]} \circ \varphi^{[1]}(x).$$

This compositional function is known as *forward propagation.*

2.1   Backpropagation

Since we wish to optimize our model with respect to our parameter $W^{[\ell]}$ and $b^{[\ell]}$, we consider a generic loss function $\mathbb{L} : \mathbb{R}^{s_2} \times \mathbb{R}^{s_2} \to \mathbb{R}$, $\mathbb{L}(\hat{y}, y)$, and by acknowledging the potential abuse of notation, we assume y is fixed, and consider the aforementioned as a function of a single-variable

$$\mathbb{L}_y : \mathbb{R}^{s_2} \to \mathbb{R}, \qquad \mathbb{L}_y(\hat{y}) = \mathbb{L}(\hat{y}, y).$$

We also define the function

$$\Phi(A, u, \xi) = A\xi + u,$$

and note that we're suppressing a dependence on the layer ℓ which only affects our domain and range of Φ (and not the actual calculations involving the derivatives). Moreover, in coordinates we see that

$$\begin{aligned}
\frac{\partial \Phi^i}{\partial A^\mu_\nu} &= \frac{\partial}{\partial A^\mu_\nu}(A^i_j \xi^j + u^i) \\
&= (\delta^i_\mu \delta^\nu_j \xi^j) \\
&= \delta^i_\mu \xi^\nu;
\end{aligned}$$

$$\frac{\partial \Phi^i}{\partial u^\mu} = \frac{\partial}{\partial u^\mu}(A^i_j \xi^j + u^i)$$
$$= \delta^i_\mu;$$

and

$$\frac{\partial \Phi^i}{\xi^\mu} = \frac{\partial}{\partial \xi^\mu}(A^i_j \xi^j + u^i)$$
$$= A^i_j \delta^j_\mu$$
$$= A^i_\mu.$$

We now define the compositional function

$$F : \mathbb{R}^{s_2 \times s_1} \times \mathbb{R}^{s_2} \times \mathbb{R}^{s_1 \times s_0} \times \mathbb{R}^{s_1} \times \mathbb{R}^{s_0} \to \mathbb{R}$$

given by

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1} \times \mathbb{1} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

We first introduce an error term $\delta^{[2]} \in \mathbb{R}^{s_2}$ defined by

$$\delta^{[2]} : = \nabla(\mathbb{L}_y \circ g^{[2]})(z^{[2]})$$
$$= (d\mathbb{L}_y \circ g^{[2]})_{z^{[2]}})^T.$$

Now we calculate the gradient $\frac{\partial F}{\partial C}$ in coordinates by

$$\delta^{[2]} = d_{z^{[2]}} F$$

$$\frac{\partial F}{\partial C^\mu_\nu} = \frac{\partial}{\partial C^\mu_\nu}\left[\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, a^{[1]})\right]$$
$$= \sum_{j=1}^{s_2} \delta^{[2]j} \frac{\partial}{\partial C^\mu_\nu}(C^j_i a^{[1]i} + c^j)$$
$$= \sum_{j=1}^{s_2} \delta^{[2]j} \delta^j_\mu a^{[1]\nu}$$
$$= \delta^{[2]}{}_\mu a^{[1]\nu}$$
$$= [a^{[1]} \delta^{[2]T}]^\nu_\mu$$

and hence that

$$\frac{\partial F}{\partial C} = \left[\frac{\partial F}{\partial C^\mu_\nu}\right]^T$$
$$= \left[\delta^{[2]}_\mu a^{[1]\nu}\right]^T$$
$$= \delta^{[2]} a^{[1]T}.$$

Moreover, we also calculate

$$\frac{\partial F}{\partial c^\mu} = \sum_{j=1}^{s_2} \delta^{[2]j} \delta_\mu^j,$$

and hence that

$$\frac{\partial F}{\partial c} = \delta^{[2]}.$$

Next we introduce another error term $\delta^{[1]} \in \mathbb{R}^{s_1}$ defined by

$$\delta^{[1]} = [dg_{z^{[1]}}^{[1]}]^T C^T \delta^{[2]}$$

with coordinates

$\delta^{[1]} = d_{z^{[1]}} F$

$$(\delta^{[1]\mu})^T = \sum_{i=1}^{s_2} \sum_{j=1}^{s_1} \delta^{[2]i} C_j^i g^{[1]\prime}(z^{[1]j}) \delta_\mu^j$$

$$= \sum_{i=1}^{s_2} \delta^{[2]i} C_\mu^i g^{[1]\prime}(z^{[1]\mu})$$

and now calculate the gradient $\frac{\partial F}{\partial B}$ in coordinates by

$$\frac{\partial F}{\partial B_\nu^\mu} = \frac{\partial}{\partial B_\nu^\mu} \left[\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b)) \right]$$

$$= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^\lambda}{\partial B_\nu^\mu}$$

$$= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \delta_\lambda^\rho g^{[1]\prime}(z^{[1]\rho}) \delta_\mu^\lambda x^\nu$$

$$= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \delta_\mu^\rho g^{[1]\prime}(z^{[1]\rho}) x^\nu$$

$$= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} C_\rho^j \delta_\mu^\rho g^{[1]\prime}(z^{[1]\rho}) x^\nu$$

$$= \sum_{j=1}^{s_2} \delta^{[2]j} C_\mu^j g^{[1]\prime}(z^{[1]\mu}) x^\nu$$

$$= \delta^{[1]}{}_\mu x^\nu$$

$$= \left[x\delta^{[1]T} \right]_\mu^\nu,$$

14

and hence that
$$\frac{\partial F}{\partial B} = \left[\frac{\partial F}{\partial B_\nu^\mu}\right]^T$$
$$= \delta^{[2]} x^T.$$

Moreover, from the above calculation, we immediately see that
$$\frac{\partial F}{\partial b^\mu} = \delta^{[1]}.$$

In summary, we've computed the following gradients
$$\frac{\partial F}{\partial W^{[2]}} = \delta^{[2]} a^{[1]T}$$
$$\frac{\partial F}{\partial b^{[2]}} = \delta^{[2]}$$
$$\frac{\partial F}{\partial W^{[1]}} = \delta^{[1]} x^T$$
$$\frac{\partial F}{\partial b^{[1]}} = \delta^{[1]},$$

where
$$\delta^{[2]} = [d(\mathbb{L}_y \circ g^{[2]})_{z^{[2]}}]^T$$
$$\delta^{[1]} = [dg_{z^{[1]}}^{[1]}]^T C^T \delta^{[2]}.$$

Finally, we recall that our cost function $\mathbb{J}$ is the average sum of our loss function $\mathbb{L}$ over our training set, we get that

$$\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) = \frac{1}{n} \sum_{j=1}^n F(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}, x_j),$$

and hence that
$$\frac{\partial \mathbb{J}}{\partial W^{[2]}} = \frac{1}{n} \sum_{j=1}^n \delta^{[2]}{}_j a^{[1]}{}_j{}^T = \frac{1}{n} \delta^{[2]} a^{[1]T}$$
$$\frac{\partial \mathbb{J}}{\partial b^{[2]}} = \frac{1}{n} \sum_{j=1}^n \delta^{[2]}{}_j$$
$$\frac{\partial \mathbb{J}}{\partial W^{[1]}} = \frac{1}{n} \sum_{j=1}^n \delta^{[1]}{}_j x_j^T = \frac{1}{n} \delta^{[1]} x^T$$
$$\frac{\partial \mathbb{J}}{\partial b^{[1]}} = \frac{1}{n} \sum_{j=1}^n \delta^{[1]}{}_j$$

15

2.2 Activation Functions

There are mainly only a handful of activating functions we consider for our non-linearity conditions.

2.2.1 The Sigmoid Function

We have the sigmoid function $\sigma(z)$ given by

$$\sigma : \mathbb{R} \to (0,1), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

We note that since

$$1 - \sigma(z) = 1 - \frac{1}{1+e^{-z}}$$
$$= \frac{e^{-z}}{1+e^{-z}}$$

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2}$$
$$= \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}}$$
$$= \sigma(z)(1-\sigma(z))$$

Moreover, suppose that $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of σ from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\sigma(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$dg_z(v) = \frac{d}{dt}\bigg|_{t=0} g(z+tv)$$
$$= \frac{d}{dt}\bigg|_{t=0} (\sigma(z^i + tv^i))$$
$$= (\sigma'(z^i)v^i)$$
$$= (\sigma(z^i)(1-\sigma(z^i))v^i)$$
$$= g(z) \odot (1-g(z)) \odot v,$$

where $\odot$ represents the Hadamard product (or component-wise multiplication); or rather, as as a matrix in $\mathbb{R}^{m \times m}$,

$$[dg_z]^\mu_\nu = \delta^\mu_\nu \sigma(z^\mu)(1-\sigma(z^\mu)).$$

2.2.2 The Hyperbolic Tangent Function

We have the hyperbolic tangent function $\tanh(z)$ given by

$$\tanh : \mathbb{R} \to (-1, 1), \qquad \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

We then calculate

$$
\begin{aligned}
\tanh'(z) &= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\
&= \frac{(e^z + e^{-z})^2}{(e^z + e^{-z})^2} - \frac{e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\
&= 1 - \tanh^2(z).
\end{aligned}
$$

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of tanh from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\tanh(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$
\begin{aligned}
dg_z(v) &= [\tanh'(z^i)] \odot [v^i] \\
&= [1 - \tanh^2(z^i)] \odot [v^i] \\
&= \delta^i_j (1 - \tanh^2(z^i)) v^j.
\end{aligned}
$$

2.2.3 The Rectified Linear Unit Function

We have the leaky-ReLU function $\text{ReLU}(z; \beta)$ given by

$$\text{ReLU} : \mathbb{R} \to \mathbb{R}, \qquad \text{ReLU}(z; \beta) = \max\{\beta z, z\},$$

for some $\beta > 0$ (typically chosen very small).

We have the rectified linear unit function $\text{ReLU}(z)$ given by setting $\beta = 0$ in the leaky-ReLu function, i.e.,

$$\text{ReLU} : \mathbb{R} \to [0, \infty), \qquad \text{ReLU}(z) = \text{ReLU}(z; \beta = 0) = \max\{0, z\}.$$

We then calculate

$$
\begin{aligned}
\text{ReLU}'(z; \beta) &= \begin{cases} \beta & z < 0 \\ 1 & z \geq 0 \end{cases} \\
&= \beta \chi_{(-\infty, 0)}(z) + \chi_{[0, \infty)}(z),
\end{aligned}
$$

where

$$\chi_A(z) = \begin{cases} 1 & z \in A \\ 0 & z \notin A \end{cases},$$

is the indicator function.

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of ReLU from $\mathbb{R}$ to $\mathbb{R}^m$. Then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = \text{ReLU}(z^i; \beta),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned} dg_z(v) &= [\text{ReLU}'(z^i; \beta)] \odot [v^i] \\ &= \delta^i_j(\beta\chi_{(-\infty,0)}(z^i) + \chi_{[0,\infty)}(z^i))v^j. \end{aligned}$$

2.2.4 The Softmax Function

We finally have the softmax function $\text{softmax}(z)$ given by

$$\text{softmax} : \mathbb{R}^m \to \mathbb{R}^m, \qquad \text{softmax}(z) = \frac{1}{\sum_{j=1}^m e^{z^j}} \begin{pmatrix} e^{z^1} \\ e^{z^2} \\ \vdots \\ e^{z^m} \end{pmatrix},$$

which we typically use on our outer-layer to obtain a probability distribution over our predicted labels. We then calculate for $z = (z^1, ..., z^m) \in \mathbb{R}^m$ that $d(\text{softmax})_z : T_z\mathbb{R}^m \to T_{\text{softmax}(z)}\mathbb{R}^m$

$$\begin{aligned} d(\text{softmax})_z(v) &= \frac{d}{dt}\Big|_{t=0} \text{softmax}(z + tv) \\ &= \frac{d}{dt}\Big|_{t=0} \frac{1}{\sum_{j=1}^m e^{z^j+tv^j}} \begin{pmatrix} e^{z^1+tv^1} \\ e^{z^2+tv^2} \\ \vdots \\ e^{z^m+tv^m} \end{pmatrix} \\ &= \frac{-1}{\left(\sum_{j=1}^m e^{z^j}\right)^2} \left(\sum_{j=1}^m e^{z^j}v^j\right) \begin{pmatrix} e^{z^1} \\ \vdots \\ e^{z^m} \end{pmatrix} + \frac{1}{\sum_{j=1}^m e^{z^j}} \begin{pmatrix} e^{z^1}v^1 \\ \vdots \\ e^{z^m}v^m \end{pmatrix} \\ &= -\langle\text{softmax}(z), v\rangle\,\text{softmax}(z) + \text{softmax}(z) \odot v, \end{aligned}$$

or rather in coordinates

$$[d(\text{softmax})_z]^i_j = S^i(\delta^i_j + \delta_{\rho j}S^\rho),$$

where

$$S^\mu = x^\mu \circ \text{softmax}(z).$$

2.3 Binary Classification - An Example

We return the network given by

$$
\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\left[z^{[2]}\right] \xrightarrow{g^{[2]}} \left[a^{[2]}\right]}_{\text{Layer 2}} \xrightarrow{=} \hat{y},
$$

and show how such a model would be trained using python below. We assume layer-2 has the sigmoid function (since it's binary classification) as an activator and our hidden layer has the ReLU function as activators.

We note that $s_2 = 1$ since we're dealing with a single activator in this layer, and

$$a^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]}),$$

with

$$d(g^{[2]})_{z^{[2]}} = \sigma'(z^{[2]}) = \sigma(z^{[2]})(1 - \sigma(z^{[2]})) = a^{[2]}(1 - a^{[2]}).$$

In layer-1, we have that

$$a^{[1]} = g^{[1]}(z^{[1]}) = \text{ReLU}(z^{[1]}),$$

with

$$d(g^{[1]})_{z^{[1]}} = \left[\delta^\mu_\nu \chi_{[0,\infty)}(z^{[1]\mu})\right]^\mu_\nu.$$

Finally, we choose our loss function $\mathbb{L}(\hat{y}, y)$ to be the log-loss function (since we're using the sigmoid activator on the outer-layer), i.e.,

$$\mathbb{L}(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y}),$$

or rather

$$\mathbb{L}(x, y) = -y\log(a^{[2]}) - (1 - y)\log(1 - a^{[2]}).$$

We then have the cost function $\mathbb{J}$ given by

$$\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) = \frac{-1}{n} \sum_{j=1}^{n} \left(y_j \log(a^{[2]}{}_j) + (1 - y_j) \log(1 - a^{[2]}{}_j) \right)$$

$$= \frac{-1}{n} \left(\langle y, \log(a^{[2]}) \rangle + \langle 1 - y, \log(1 - a^{[2]}) \rangle \right)$$

Moreover, when using backpropagation, we see that

$$\delta^{[2]T}{}_j = d(\mathbb{L}_{y_j})_{a^{[2]}} \cdot d(g^{[2]})_{z^{[2]}{}_j}$$

$$= \left(-\frac{y_j}{a^{[2]}{}_j} + \frac{1 - y_j}{1 - a^{[2]}{}_j} \right) \cdot (a^{[2]}{}_j(1 - a^{[2]}{}_j))$$

$$= a^{[2]}{}_j - y_j,$$

or rather

$$\delta^{[2]} = a^{[2]} - y.$$

Similarly, we compute

$$\delta^{[1]T}{}_j = \delta^{[2]T}{}_j W^{[2]} [dg^{[1]}_{z^{[1]}{}_j}]$$

$$= \delta^{[2]T}{}_j W^{[2]} [\delta^{\mu}_{\nu} \cdot \chi_{[0,\infty)}(z^{[1]\mu}{}_j)]$$

2.3.1   Random Initialization

In the section that follows, we see that to begin gradient descent for a shallow neural network, we initialize our parameters $b^{[\ell]}$ to be 0, but choose an arbitrarily small, but nonzero initialization for $W^{[\ell]}$. Let's see why we choose $W^{[\ell]}$ to be nonzero. Indeed, suppose we initialize with $b^{[\ell]} = 0$ and $W^{[\ell]} = 0$. Then we see that

$$\delta^{[1]T} = \delta^{[2]} W^{[2]} dg^{[1]}_{z^{[1]}} = 0,$$

and so

$$\frac{\partial \mathbb{J}}{\partial W^{[1]}} = \frac{1}{n} \delta^{[1]} x^T = 0.$$

Then we conclude that our parameter $W^{[1]}$ remains at 0 during every iteration which is enough reason to not initialize $W^{[2]}$ at 0. Similarly, since

$$a^{[1]} = \tanh(W^{[1]}x + b^{[1]}) = \tanh(0) = 0,$$

we reach a similar conclusion about $W^{[1]}$ and $W^{[2]}$, respectively.

2.3.2   Vectorization in Python

```python
1  import copy
2
3  import numpy as np
4
5  # Activator functions
6
7  def sigmoid(z):
8      """
9      Parameters
10     ----------
11     z : array_like
12
13     Returns
14     -------
15     sigma : array_like
16         The value of the sigmoid function evaluated at z
17     ds : array_like
18         The differential of the sigmoid function evaluate at z
19     """
20     # Compute value of sigmoid
21     sigma = (1 / (1 + np.exp(-z)))
22     # Compute differential of sigmoid
23     ds = sigma * (1 - sigma)
24     return sigma, ds
25
26 # Preliminary functions for our model
27 def layer_shapes(x, y, hidden_layer_size):
28     """
29     Parameters
30     ----------
31     x : array_like
32         x.shape = (m_x, n)
33     y : array_like
34         y.shape = (m_y, n)
35     hidden_layer_size : int
36         The number nodes in the hidden layer
37     Returns
38     -------
39     n : int
40         The number of training examples
41     m_x : int
42         The number of input features
43     m_h : The number of nodes in the hidden layer
44     m_y : The number of nodes in the output layer
45     """
```

```
46      m_x, n = x.shape
47      assert(y.shape[1] == n)
48      m_y = y.shape[0]
49      m_h = hidden_layer_size
50      return n, m_x, m_h, m_y
51
52
53
54  def initialize_parameters(m_x, m_h, m_y):
55      """
56      Parameters
57      ----------
58      m_x : int
59          The number of input features
60      m_h : int
61          The number of nodes in the hidden layer
62      m_y : int
63          The number of nodes in the output layer
64
65      Returns
66      -------
67      params : Dict
68          w1 : array_like
69              w1.shape = (m_h, m_x)
70          b1 : array_like
71              b1.shape = (m_h, 1)
72          w2 : array_like
73              w2.shape= (m_y, m_h)
74          b2 : array_like
75              b2.shape = (m_y, 1)
76      """
77      w1 = np.random.randn(m_h, m_x) * 0.01
78      b1 = np.zeros((m_h, 1))
79      w2 = np.random.randn(m_y, m_h) * 0.01
80      b2 = np.zeros((m_y, 1))
81
82      params = {'w1' : w1,
83                'b1' : b1,
84                'w2' : w2,
85                'b2' : b2}
86
87      return params
88
89  def forward_propagation(x, params):
90      """
91      Parameters
92      ----------
```

```
93      x : array_like
94          x.shape = (m_x, n)
95      params : Dict
96          params['w1'] : array_like
97              w1.shape = (m_h, m_x)
98          params['b1'] : array_like
99              b1.shape = (m_h, 1)
100         params['w2'] : array_like
101             w2.shape = (m_y, m_h)
102         params['b2'] : array_like
103             b2.shape = (m_y, 1)
104     Returns
105     -------
106     a2 : array_like
107         a2.shape = (m_y, n)
108     cache : Dict
109         cache['z1'] : array_like
110             z1.shape = (m_h, n)
111         cache['a1'] : array_like
112             a1.shape = (m_h, n)
113         cache['z2'] : array_like
114             z2.shape = (m_y, n)
115         cache['a2'] = a2
116     """
117
118     # Retrieve parameters
119     w1 = params['w1']
120     b1 = params['b1']
121     w2 = params['w2']
122     b2 = params['b2']
123
124     # Auxiliary computations
125     z1 = w1 @ x + b1
126     a1 = np.tanh(z1)
127     z2 = w2 @ a1 + b2
128     a2 = sigmoid(z2)
129
130     assert(a1.shape == (w1.shape[0], x.shape[1]))
131     assert(a2.shape == (w2.shape[0], a1.shape[1]))
132
133     cache = {'z1' : z1,
134              'a1' : a1,
135              'z2' : z2,
136              'a2' : a2}
137
138     return a2, cache
139
```

```python
140  def compute_cost(a2, y):
141      """
142      Parameters
143      ----------
144      a2 : array_like
145          a2.shape = (m_y, n)
146      y : array_like
147          y.shape = (m_y, n)
148      Returns
149      -------
150      cost : float
151          The cost evaluated at y and a2
152      """
153      n = y.shape[1]
154      cost = (-1 / n) * (np.sum(y * np.log(a2)) + np.sum((1 - y) * np.log(1 - a2)))
155      cost = float(np.squeeze(cost))  # Makes sure we return a float
156
157      return cost
158
159  def backward_propagation(params, cache, x, y):
160      """
161      Parameters
162      ----------
163      params : Dict
164          params['w2'] : array_like
165              w2.shape = (m_y, m_h)
166          params['b2'] : array_like
167              b2.shape = (m_y, 1)
168          params['w1'] : array_like
169              w1.shape = (m_h, m_x)
170          params['b1'] : array_like
171              b1.shape = (m_h, 1)
172      cache : Dict
173          cache['z1'] : array_like
174              z1.shape = (m_h, n)
175          cache['a1'] : array_like
176              a1.shape = (m_h, n)
177          cache['z2'] : array_like
178              z2.shape = (m_y, n)
179          cache['a2'] = a2
180      x : array_like
181          x.shape = (m_x, n)
182      y : array_like
183          y.shape = (m_y, n)
184      Returns
185      -------
186      grads : Dict
```

```
187        grads['dw2'] : array_like
188            dw2.shape = (m_y, m_h)
189        grads['db2'] : array_like
190            db2.shape = (m_y, 1)
191        grads['dw1'] : array_like
192            dw1.shape = (m_h, m_x)
193        grads['db1'] : array_like
194            db1.shape = (m_h, 1)
195    """
196    # Retrieve parameters
197    w1 = params['w1']
198    w2 = params['w2']
199
200    # Set dimensional constants
201    m_x, n = x.shape
202    m_y, m_h = w2.shape
203
204    # Retrieve node outputs
205    a1 = cache['a1']
206    a2 = cache['a2']
207
208    # Auxiliary Computations
209    delta2 = a2 - y
210    assert(delta2.shape ==(m_y, n))
211    d_tanh = 1 - (a1 * a1)
212    assert(d_tanh.shape == (m_h, n))
213    delta1 = (w2.T @ delta1) * d_tanh
214    assert(delta1.shape == (m_h, n))
215
216    # Gradient computations
217    dw2 = (1 / n) * delta2 @ a1.T
218    db2 = (1 / n) * np.sum(delta2, axis=1, keepdims=True)
219    dw1 = (1 / n) * delta1 @ x.T
220    db1 = (1 / n) * np.sum(delta1, axis=1, keepdims=True)
221
222    # Combine and return dict
223    grads = {'dw2' : dw2,
224             'db2' : db2,
225             'dw1' : dw1,
226             'db1' : db1}
227    return grads
228
229 def update_parameters(params, grads, learning_rate=1.2):
230    """
231    Parameters
232    ----------
233    params : Dict
```

```
234          params['w2'] : array_like
235              w2.shape = (m_y, m_h)
236          params['b2'] : array_like
237              b2.shape = (m_y, 1)
238          params['w1'] : array_like
239              w1.shape = (m_h, m_x)
240          params['b1'] : array_like
241              b1.shape = (m_h, 1)
242      grads : Dict
243          grads['dw2'] : array_like
244              dw2.shape = (m_y, m_h)
245          grads['db2'] : array_like
246              db2.shape = (m_y, 1)
247          grads['dw1'] : array_like
248              dw1.shape = (m_h, m_x)
249          grads['db1'] : array_like
250              db1.shape = (m_h, 1)
251      learning_rate : float
252          Default = 1.2
253      Returns
254      -------
255      params : Dict
256          params['w2'] : array_like
257              w2.shape = (m_y, m_h)
258          params['b2'] : array_like
259              b2.shape = (m_y, 1)
260          params['w1'] : array_like
261              w1.shape = (m_h, m_x)
262          params['b1'] : array_like
263              b1.shape = (m_h, 1)
264      """
265      # Retrieve parameters
266      w2 = copy.deepcopy(params['w2'])
267      b2 = params['b2']
268      w1 = copy.deepcopy(params['w1'])
269      b1 = params['b1']
270
271      # Retrieve gradients
272      dw2 = grads['dw2']
273      db2 = grads['db2']
274      dw1 = grads['dw1']
275      db1 = grads['db1']
276
277      # Perform update
278      w2 = w2 - learning_rate * dw2
279      b2 = b2 - learning_rate * db2
280      w1 = w1 - learning_rate * dw1
```

```python
281     b1 = b1 - learning_rate * db1
282
283     # Combine and return dict
284     params = {'w2' : w2,
285               'b2' : b2,
286               'w1' : w1,
287               'b1' : b1}
288     return params
289
290
291 # The main neural network training model
292 def model(x, y, num_hidden_layer, num_iters=10000, print_cost=False):
293     """
294     Parameters
295     ----------
296     x : array_like
297         x.shape = (m_x, n)
298     y : array_like
299         y.shape = (m_y. n)
300     num_hidden_layer : int
301         Number of nodes in the single hidden layer
302     num_iters : int
303         Number of iterations with which our model performs gradient descent
304     print_cost : Boolean
305         If True, print the cost every 1000 iterations
306     Returns
307     -------
308     params : Dict
309         params['w2'] : array_like
310             w2.shape = (m_y, m_h)
311         params['b2'] : array_like
312             b2.shape = (m_y, 1)
313         params['w1'] : array_like
314             w1.shape = (m_h, m_x)
315         params['b1'] : array_like
316             b1.shape = (m_h, 1)
317     """
318     # Set dimensional constants
319     n, m_x, m_h, m_y = layer_shapes(x, y, num_hidden_layer)
320     # initialize parameters
321     params = initialize_parameters(m_x, m_h, m_y)
322
323     # main loop for gradient descent
324     for i in range(num_iters):
325         a2, cache = forward_propagation(X, params)
326         cost = compute_cost(a2, y)
327         grads = backward_propagation(params, cache, x, y)
```

```python
328         params = update_parameters(params, grads)
329
330         if print_cost and i % 1000 == 0:
331             print(f'Cost after iteration {i}: {cost}')
332
333     return params
334
335 # Using our model to obtain predictions
336 def predict(params, x):
337     """
338     Parameters
339     ----------
340     params : Dict
341         params['w2'] : array_like
342             w2.shape = (m_y, m_h)
343         params['b2'] : array_like
344             b2.shape = (m_y, 1)
345         params['w1'] : array_like
346             w1.shape = (m_h, m_x)
347         params['b1'] : array_like
348             b1.shape = (m_h, 1)
349     x : array_like
350         x.shape = (m_x, n)
351
352     Returns
353     -------
354     predictions : array_like
355         predictions.shape = (m_y, n)
356     """
357     a2, _ = forward_propagation(x, params)
358     predictions = np.zeros(a2.shape)
359     predictions[~(a2 < 0.5)] = 1
360
361     return predictions
```

3 Deep Neural Networks

In this section we discuss a general "deep" neural network, which consist of L layers. That is, we have a network of the form:

$$\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix} \xrightarrow{\varphi^{[1]}} \begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix} \xrightarrow{\varphi^{[2]}} \begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]s_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]s_2} \end{bmatrix} \xrightarrow{\varphi^{[3]}} \cdots$$

Layer 0   Layer 1   Layer 2

$$\cdots \xrightarrow{\varphi^{[L-1]}} \begin{bmatrix} z^{[L-1]1} \\ \vdots \\ z^{[L-1]s_{L-1}} \end{bmatrix} \xrightarrow{g^{[L-1]}} \begin{bmatrix} a^{[L-1]1} \\ \vdots \\ a^{[L-1]s_{L-1}} \end{bmatrix} \xrightarrow{\varphi^{[L]}} \begin{bmatrix} z^{[L]1} \\ \vdots \\ z^{[L]s_L} \end{bmatrix} \xrightarrow{g^{[L]}} \begin{bmatrix} a^{[L]1} \\ \vdots \\ a^{[L]s_L} \end{bmatrix} \xrightarrow{=} \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{s_L} \end{bmatrix},$$

Layer $L-1$ Layer L

where

$$s_\ell := \text{ the number of nodes in layer-}\ell,$$

$$\varphi^{[\ell]} : \mathbb{R}^{s_{\ell-1}} \to \mathbb{R}^{s_\ell}, \qquad \varphi^{[\ell]}(\xi) = W^{[\ell]}\xi + b^{[\ell]}, \qquad W^{[\ell]} \in \mathbb{R}^{s_\ell \times s_{\ell-1}}, b \in \mathbb{R}^{s_\ell},$$

and

$$g^{[\ell]} : \mathbb{R}^{s_\ell} \to \mathbb{R}^{s_\ell},$$

is a broadcasted activation function determined by the layer-ℓ.

As with a shallow network, our functional composition to obtain $a^{[L]}$ is known as forward propagation.

3.1 Backpropagation

As the general derivation for backpropagation can be easily (if not tediously) generalized from Section 2.1 using induction, we give the general outline for computational purposes.

Let $\mathbb{L} : \mathbb{R}^{s_L} \times \mathbb{R}^{s_L} \to \mathbb{R}$ be a generic loss function, and suppose our cost function is given by the usual

$$\mathbb{J}(W, b) = \frac{1}{n} \sum_{j=1}^{n} \mathbb{L}(\hat{y}_j, y_j).$$

Then from previous computations, we have the following gradients for any

$\ell \in \{1, 2, ..., L\}$, that

$$\frac{\partial \mathbb{J}}{\partial W^{[\ell]}} = \frac{1}{n} \delta^{[\ell]} a^{[\ell-1]T}$$

$$\frac{\partial \mathbb{J}}{\partial b^{[\ell]}} = \frac{1}{n} \sum_{j=1}^{n} \delta^{[\ell]}_{j}$$

where we impose the notation of

$$a^{[0]} := x.$$

So we need only give a full characterization of $\delta^{[\ell]}$.. To this end, we define recursively starting at layer-L by

$$\delta^{[L]T} := d(\mathbb{L}_y)_{a^{[L]}} \cdot dg^{[L]}_{z^{[L]}},$$
$$\delta^{[L-1]T} := \delta^{[L]T} \cdot W^{[L]} \cdot dg^{[L-1]}_{z^{[L-1]}},$$
$$\vdots$$
$$\delta^{[\ell]T} := \delta^{[\ell+1]T} W^{[\ell+1]} dg^{[\ell]}_{z^{[\ell]}},$$
$$\vdots$$
$$\delta^{[1]T} := \delta^{[2]T} W^{[2]} dg^{[1]}_{z^{[1]}},$$

as desired.

3.1.1   Vectorization in Python

We implement a neural network with an arbitrary number of layers and nodes, with the ReLU function as the activator on all hidden nodes and the sigmoid function on the output layer for binary classification with the log-loss function.

```
1  import copy
2
3  import numpy as np
4
5  ## Activator functions
6  def relu(z, beta=0.0):
7      """
8      Parameters
9      ----------
10     z : array_like
```

```python
11     beta : float
12
13     Returns
14     -------
15     r : array_like
16         The ReLU function when beta=0, the leaky-ReLU otherwise.
17     dr : array_like
18         The differential of the ReLU function
19     """
20     # Change scalar to array if needed
21     z = np.array(z)
22     # Compute value of ReLU(z)
23     r = np.maximum(z, beta * z)
24     # Compute differential ReLU'(z)
25     dr = (~(z < 0)) * 1
26     return r, dr
27
28 def sigmoid(z):
29     """
30     Parameters
31     ----------
32     z : array_like
33
34     Returns
35     -------
36     sigma : array_like
37         The value of the sigmoid function evaluated at z
38     ds : array_like
39         The differential of the sigmoid function evaluate at z
40     """
41     # Compute value of sigmoid
42     sigma = (1 / (1 + np.exp(-z)))
43     # Compute differential of sigmoid
44     ds = sigma * (1 - sigma)
45     return sigma, ds
46
47
48 ## Auxiliary functions for model composition
49 def dim_retrieval(x, y, hidden_sizes):
50     """
51     Parameters
52     ----------
53     x : array_like
54         x.shape = (layers[0], n)
55     y : array_like
56         y.shape = (layers[L], n)
57     hidden_sizes : List[int]
```

```
58          The number nodes layer i = hidden_sizes[i-1]
59      Returns
60      -------
61      n : int
62          The number of training examples
63      layers : List
64          layer[l] = # nodes in layer l
65
66      """
67      m, n = x.shape
68      assert(y.shape[1] == n)
69      K = y.shape[0]
70      layers = [m]
71      layers.extend(hidden_sizes)
72      layers.append(K)
73
74      return n, layers
75
76  def initialize_parameters(layers):
77      """
78      Parameters
79      ----------
80      layers : List[int]
81          layers[l] = # nodes in layer l
82      Returns
83      -------
84      params : Dict[Dict]
85          w[l] : array_like
86              dwl.shape = (layers[l], layers[l-1])
87          b[l] : array_like
88              dbl.shape = (layers[l], 1)
89      """
90      w = {}
91      b = {}
92      for l in range(1, len(layers)):
93          w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
94          b[l] = np.zeros((layers[l], 1))
95      params = {'w' : w, 'b' : b}
96      return params
97
98  def forward_propagation(params, x, activators):
99      """
100     Parameters
101     ----------
102     params : Dict[Dict]
103         params['w'][l] : array_like
104             wl.shape = (layers[l], layers[l-1])
```

```
105        params['b'][l] : array_like
106            bl.shape = (layers[l], 1)
107    x : array_like
108        x.shape = (layers[0] n)
109    activators : List[function]
110        activators[l] = activation function of layer l+1
111    Returns
112    -------
113    cache : Dict[Dict]
114        cache['z'][l] : array_like
115            z[l].shape = (layers[l], n)
116        cache['a'][l] : array_like
117            a[l].shape = (layers[l], n)
118    """
119    n = x.shape[1]
120    # Number of layers including input-layer
121    L = len(params['w']) + 1
122    a = {}
123    z = {}
124    a[0] = x
125    for l in range(1, L):
126        w = params['w'][l]
127        temp_a = a[l - 1]
128        b = params['b'][l]
129        temp_z = w @ temp_a + b
130        assert(temp_z.shape == (w.shape[0], n))
131        z[l] = temp_z
132        a[l], _ = activators[l - 1](temp_z)
133        assert(a[l].shape == temp_z.shape)
134
135    cache = {'a' : a, 'z' : z}
136    return cache
137
138 def compute_cost(cache, y):
139    """
140    Parameters
141    ----------
142    cache : Dict[Dict]
143        cache['z'][l] : array_like
144            z[l].shape = (layers[l], n)
145        cache['a'][l] : array_like
146            a[l].shape = (layers[l], n)
147    y : array_like
148        y.shape = (layers[-1], n)
149    -------
150    cost : float
151        The cost evaluated at y and aL
```

```
152         """
153         ## Retrieve parameters
154         n = y.shape[1]
155         a = cache['a']
156         L = len(a)
157         aL = a[L - 1]
158
159         cost = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
160         cost = float(np.squeeze(cost))
161
162         return cost
163
164 def backward_propagation(params, cache, activators, x, y):
165         """
166         Parameters
167         ----------
168         params : Dict
169             params['w'][l] : array_like
170                 w[l].shape = (layers[l], layers[l-1])
171             params['b'][l] : array_like
172                 b[l].shape = (layers[l], 1)
173         cache : Dict
174             cache['a'][l] : array_like
175                 a[l].shape = (layers[l], n)
176             cache['z'][l] : array_like
177                 z[l].shape = (layers[l], n)
178         activators : List[function]
179             activators[l] = activation function of layer l+1
180         x : array_like
181             x.shape = (layers[0], n)
182         y : array_like
183             y.shape = (layers[-1], n)
184         Returns
185         -------
186         grads : Dict[Dict]
187             grads['dw'][l] : array_like
188                 dw[l].shape = w[l].shape
189             grads['db'][l] : array_like
190                 db[l].shape = b[l].shape
191         """
192         ## Retrieve parameters
193         a = cache['a']
194         z = cache['z']
195         w = params['w']
196         n = x.shape[1]
197         L = len(a) - 1
198
```

```
199     ## Compute deltas
200     delta = {}
201     delta[L] = a[L] - y
202     for l in range(L-1, 0, -1):
203         _, dg = activators[l](z[l])
204         delta[l] = (delta[l+1].T @ w[l+1]).T * dg
205         assert(delta[l].shape == (w[l].shape[0], n))
206
207     ## Compute gradients
208     dw = {}
209     db = {}
210     for l in range(1, L + 1):
211         db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
212         assert(db[l].shape == (w[l].shape[0], 1))
213         dw[l] = (1 / n) * delta[l] @ a[l - 1].T
214         assert(dw[l].shape == w[l].shape)
215     grads ={'dw' : dw, 'db' : db}
216     return grads
217
218 def update_parameters(params, grads, learning_rate=0.01):
219     """
220     Parameters
221     ----------
222     params : Dict[Dict]
223         params['w'][l] : array_like
224             w[l].shape = (layers[l], layers[l-1])
225         params['b'][l] : array_like
226             b[l].shape = (layers[l], 1)
227     grads : Dict[Dict]
228         grads['dw'][l] : array_like
229             dw[l].shape = w[l].shape
230         grads['db'][l] : array_like
231             db[l].shape = b[l].shape
232     learning_rate : float
233         Default: 0.01
234         The learning rate for gradient descent
235
236     Returns
237     -------
238     params : Dict[Dict]
239         params['w'][l] : array_like
240             w[l].shape = (layers[l], layers[l-1])
241         params['b'][l] : array_like
242             b[l].shape = (layers[l], 1)
243     """
244     ## Retrieve parameters
245     w = copy.deepcopy(params['w'])
```

```
246    b = copy.deepcopy(params['b'])
247    L = len(w)
248
249    ## Retrieve gradients
250    dw = grads['dw']
251    db = grads['db']
252
253    ## Perform update
254    for l in range(1, L + 1):
255        w[l] = w[l] - learning_rate * dw[l]
256        b[l] = b[l] - learning_rate * db[l]
257
258    params = {'w' : w, 'b' : b}
259    return params
260
261
262 ## The main model for training our parameters
263 def model(x, y, hidden_layer_sizes, activators, num_iters=10000, print_cost=False):
264    """
265    Parameters
266    ----------
267    x : array_like
268        x.shape = (layers[0], n)
269    y : array_like
270        y.shape = (layers[-1], n)
271    hidden_layer_sizes : List[int]
272        The number nodes layer l = hidden_layer_sizes[l-1]
273    activators : List[function]
274        activators[l] = activation function of layer l+1
275    num_iters : int
276        Number of iterations with which our model performs gradient descent
277    print_cost : Boolean
278        If True, print the cost every 1000 iterations
279    Returns
280    -------
281    params : Dict[Dict]
282        params['w'][l] : array_like
283            w[l].shape = (layers[l], layers[l-1])
284        params['b'][l] : array_like
285            b[l].shape = (layers[l], 1)
286    cost : float
287        The final cost value for the optimized parameters returned
288    """
289    ## Set dimensions and Initialize parameters
290    n, layers = dim_retrieval(x, y, hidden_layer_sizes)
291    params = initialize_parameters(layers)
292
```

```python
293    ## main loop
294    for i in range(num_iters):
295        cache = forward_propagation(params, x, activators)
296        cost = compute_cost(cache, y)
297        grads = backward_propagation(params, cache, activators, x, y)
298        params = update_parameters(params, grads, 0.1)
299
300        if print_cost and i % 1000 == 0:
301            print(f'Cost after iteration {i}: {cost}')
302
303    return params, cost
```

Part II

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization

4 Training, Development and Test Sets