

# Neural Networks

Matt R

February 24, 2022

## Contents

<b>1</b>	<b>Logistic Regression</b>	<b>2</b>
1.1	The Gradient . . . . .	3
1.1.1	Vectorization in Python . . . . .	4
<b>2</b>	<b>Neural Networks: A Single Hidden Layer</b>	<b>9</b>
2.1	Backpropagation . . . . .	10

# 1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose 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  $\sigma : \mathbb{R} \rightarrow (0, 1)$  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

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

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  $0 \leq a \leq 1$ , 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  $(0, 1) \rightarrow (-\infty, 0)$ . This leads us to define our log-loss function

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

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

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

## 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  $\mathbb{L} : (0, 1) \times \mathbb{R} \rightarrow \mathbb{R}$ ,

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

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

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

Then by the chain rule, we have that

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

Composition turns into matrix multiplication in the tangent space.

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

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

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

$$\begin{aligned} \partial_w \mathbb{J}(w, b) &= \frac{1}{n} \sum_{j=1}^n (a_j - y_j) x_j \\ &= \frac{1}{n} ((a - y) \cdot x^T)^T \\ &= \frac{1}{n} x \cdot (a - y)^T \end{aligned}$$

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.

```

1 import copy
2
3 import numpy as np
4
5 def sigmoid(z):
6     """
7     Parameters
8     -----
9     z : array_like
10
11     Returns
12     -----
13     sigma : array_like
14     """
15
16     sigma = (1 / (1 + np.exp(-z)))
17     return sigma
18

```

```

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 respect 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=True)
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

```

```

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  $y \in \{0, 1\}^n$ . Usually with logistic regression we have the following type of structure:

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

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. 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:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{g^{[1]}} \underbrace{\begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \\ \vdots \\ z^{[2]s_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{g^{[2]}} \underbrace{\begin{bmatrix} a^{[2]} \\ \vdots \\ a^{[2]s_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

where

$$\begin{aligned} \varphi^{[1]} : \mathbb{R}^{s_0} &\rightarrow \mathbb{R}^{s_1}, & \varphi^{[1]}(x) &= W^{[1]}x + b^{[1]}, \\ \varphi^{[2]} : \mathbb{R}^{s_1} &\rightarrow \mathbb{R}, & \varphi^{[2]}(x) &= W^{[2]}x + b^{[2]}, \end{aligned}$$

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  $g : \mathbb{R} \rightarrow \mathbb{R}$  is any function. Then we say  $\bar{g} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$  is the **broadcast** of  $g$  if

$$\begin{aligned} \bar{g}(A) &= \bar{g}(A_j^i e_i^j) \\ &= g(A_j^i) e_i^j, \end{aligned}$$

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

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:

$$\begin{aligned} \varphi^{[1]} : \mathbb{R}^{s_0} &\rightarrow \mathbb{R}^{s_1}, & d\varphi^{[1]} : T\mathbb{R}^{s_0} &\rightarrow T\mathbb{R}^{s_1}, \\ z^{[1]} = \varphi^{[1]}(x) &= W^{[1]}x + b^{[1]}, & d\varphi_x^{[1]}(v) &= W^{[1]}v; \end{aligned}$$

$$\begin{aligned} g^{[1]} : \mathbb{R}^{s_1} &\rightarrow \mathbb{R}^{s_1}, & dg^{[1]} : T\mathbb{R}^{s_1} &\rightarrow T\mathbb{R}^{s_1}, \\ a^{[1]} = g^{[1]}(z^{[1]}), & & \frac{\partial a^{[1]\mu}}{\partial z^{[1]\nu}} &= \delta_\nu^\mu g^{[1]'}(z^{[1]\mu}); \end{aligned}$$

$$\begin{aligned} \varphi^{[2]} : \mathbb{R}^{s_1} &\rightarrow \mathbb{R}^{s_2}, & d\varphi^{[2]} : T\mathbb{R}^{s_1} &\rightarrow T\mathbb{R}^{s_2}, \\ z^{[2]} = \varphi^{[2]}(a^{[1]}) &= W^{[2]}a^{[1]} + b^{[2]}, & d\varphi_{a^{[1]}}^{[2]}(v) &= W^{[2]}v; \end{aligned}$$

$$\begin{aligned} g^{[2]} : \mathbb{R}^{s_2} &\rightarrow \mathbb{R}^{s_2}, & dg^{[2]} : T\mathbb{R}^{s_2} &\rightarrow T\mathbb{R}^{s_2}, \\ a^{[2]} = g^{[2]}(z^{[2]}), & & \frac{\partial a^{[2]\mu}}{\partial z^{[2]\nu}} &= \delta_\nu^\mu g^{[2]'}(z^{[2]\mu}). \end{aligned}$$

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  $\mathbb{L} : \mathbb{R}^{s_2} \times \mathbb{R}^{s_2} \rightarrow \mathbb{R}$ ,  $\mathbb{L}(\hat{y}, y)$ , and by acknowledging the potential abuse of notation, we assume  $y$  is fixed, and consider the aforementioned as a function of a single-variable

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

We also define the function

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

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

$$\begin{aligned}\frac{\partial \Phi^i}{\partial A_\nu^\mu} &= \frac{\partial}{\partial A_\nu^\mu} (A_j^i \xi^j + u^i) \\ &= (\delta_\mu^i \delta_j^\nu \xi^j) \\ &= \delta_\mu^i \xi^\nu;\end{aligned}$$

$$\begin{aligned}\frac{\partial \Phi^i}{\partial u^\mu} &= \frac{\partial}{\partial u^\mu} (A_j^i \xi^j + u^i) \\ &= \delta_\mu^i;\end{aligned}$$

and

$$\begin{aligned}\frac{\partial \Phi^i}{\xi^\mu} &= \frac{\partial}{\partial \xi^\mu} (A_j^i \xi^j + u^i) \\ &= A_j^i \delta_\mu^j \\ &= A_\mu^i.\end{aligned}$$

We now define the compositional function

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

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

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

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

$$\begin{aligned}
\frac{\partial F}{\partial C_\nu^\mu} &= \frac{\partial}{\partial C_\nu^\mu} [\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, a^{[1]})] \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \frac{\partial}{\partial C_\nu^\mu} (C_i^j a^{[1]i} + c^j) \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \delta_\mu^j a^{[1]\nu} \\
&= \delta^{[2]}_\mu a^{[1]\nu} \\
&= [a^{[1]} \delta^{[2]T}]_\mu^\nu
\end{aligned}$$

and hence that

$$\begin{aligned}
\frac{\partial F}{\partial C} &= \left[ \frac{\partial F}{\partial C_\nu^\mu} \right]^T \\
&= [\delta_\mu^{[2]} a^{[1]\nu}]^T \\
&= \delta^{[2]} a^{[1]T}.
\end{aligned}$$

Moreover, we also calculate

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

and hence that

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

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

$$\begin{aligned}
\frac{\partial F}{\partial B_\nu^\mu} &= \frac{\partial}{\partial B_\nu^\mu} [\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b))] \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^\lambda}{\partial B_\nu^\mu} \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \delta_\lambda^\rho g^{[1]'}(z^{[1]\rho}) \delta_\mu^\lambda x^\nu \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} C_\rho^j \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} C_\mu^j g^{[1]'}(z^{[1]})
\end{aligned}$$