

Neural Networks

Matt R

March 7, 2022

Contents

I	Neural Networks and Deep Learning	2
1	Logistic Regression	3
1.1	The Gradient	4
1.1.1	Vectorization in Python	5
2	Neural Networks: A Single Hidden Layer	10
2.1	Backpropagation	12
2.2	Activation Functions	16
2.2.1	The Sigmoid Function	16
2.2.2	The Hyperbolic Tangent Function	17
2.2.3	The Rectified Linear Unit Function	17
2.2.4	The Softmax Function	18
2.3	Binary Classification - An Example	19
2.3.1	Random Initialization	20
2.3.2	Vectorization in Python	21
3	Deep Neural Networks	29
3.1	Backpropagation	29
3.1.1	Vectorization in Python	30
II	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization	38

4	Training, Development and Test Sets	39
4.0.1	Python Implementation	41
5	Regularization	42
5.1	(Inverted) Dropout Regularization	43
5.1.1	Python Implementation	44
5.2	Data Augmentation	50
5.3	Early Stopping	50
6	Gradients and Numerical Remarks	51
6.1	Numerical Gradient Checking	51
6.1.1	Python Implementation	52

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 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 $y \in \{0, 1\}^{1 \times n}$. 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, 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:

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

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 $G : \mathbb{R}^m \rightarrow \mathbb{R}^m$ is the **broadcast** of g from \mathbb{R} to \mathbb{R}^m if

$$\begin{aligned} G(v) &= G(v^i e_i) \\ &= g(v^i) e_i, \end{aligned}$$

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 .

castingDifferential

Lemma 2.2. Suppose $g : \mathbb{R} \rightarrow \mathbb{R}$ is any smooth function and $G : \mathbb{R}^m \rightarrow \mathbb{R}^m$ is the broadcasting of g from \mathbb{R} to \mathbb{R}^m . Then the differential $dG_z : T_z \mathbb{R}^m \rightarrow T_{G(z)} \mathbb{R}^m$ 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

$$[dG_z]_j^i = \delta_j^i g'(z^i).$$

Proof: We calculate

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

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

$$\begin{aligned} dG_z(e_j) &= [g'(z^i)] \odot e_j \\ &= g'(z^j) e_j, \end{aligned}$$

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

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

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

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

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

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

backPropDerivation

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 ℓ which only affects our domain and range of Φ (and not the actual calculations involving the derivatives). Moreover, in coordinates we see that

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

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

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

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

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

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

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

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

$$\begin{aligned} (\delta^{[1]\mu})^T &= \sum_{i=1}^{s_2} \sum_{j=1}^{s_1} \delta^{[2]i} C_j^i g^{[1]'}(z^{[1]j}) \delta_\mu^j \\ &= \sum_{i=1}^{s_2} \delta^{[2]i} C_\mu^i g^{[1]'}(z^{[1]\mu}) \end{aligned}$$

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

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

$$\begin{aligned} \frac{\partial F}{\partial B_\nu^\mu} &= \frac{\partial}{\partial B_\nu^\mu} [\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b))] \\ &= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^\lambda}{\partial B_\nu^\mu} \\ &= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{s_1} \delta_\lambda^\rho g^{[1]'}(z^{[1]\rho}) \delta_\mu^\lambda x^\nu \\ &= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\ &= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} C_\rho^j \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\ &= \sum_{j=1}^{s_2} \delta^{[2]j} C_\mu^j g^{[1]'}(z^{[1]\mu}) x^\nu \\ &= \delta^{[1]}_\mu x^\nu \\ &= [x \delta^{[1]T}]_\mu^\nu, \end{aligned}$$

and hence that

$$\begin{aligned}\frac{\partial F}{\partial B} &= \left[\frac{\partial F}{\partial B_\nu^\mu} \right]^T \\ &= \delta^{[2]} x^T.\end{aligned}$$

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

$$\begin{aligned}\frac{\partial F}{\partial W^{[2]}} &= \delta^{[2]} a^{[1]T} \\ \frac{\partial F}{\partial b^{[2]}} &= \delta^{[2]} \\ \frac{\partial F}{\partial W^{[1]}} &= \delta^{[1]} x^T \\ \frac{\partial F}{\partial b^{[1]}} &= \delta^{[1]},\end{aligned}$$

where

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

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

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial W^{[2]}} &= \frac{1}{n} \sum_{j=1}^n \delta^{[2]}_j a^{[1]}_j{}^T = \frac{1}{n} \delta^{[2]} a^{[1]T} \\ \frac{\partial \mathbb{J}}{\partial b^{[2]}} &= \frac{1}{n} \sum_{j=1}^n \delta^{[2]}_j \\ \frac{\partial \mathbb{J}}{\partial W^{[1]}} &= \frac{1}{n} \sum_{j=1}^n \delta^{[1]}_j x_j^T = \frac{1}{n} \delta^{[1]} x^T \\ \frac{\partial \mathbb{J}}{\partial b^{[1]}} &= \frac{1}{n} \sum_{j=1}^n \delta^{[1]}_j\end{aligned}$$

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

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

We note that since

$$\begin{aligned} 1 - \sigma(z) &= 1 - \frac{1}{1 + e^{-z}} \\ &= \frac{e^{-z}}{1 + e^{-z}} \end{aligned}$$

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

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

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

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

$$\begin{aligned} dg_z(v) &= \left. \frac{d}{dt} \right|_{t=0} g(z + tv) \\ &= \left. \frac{d}{dt} \right|_{t=0} (\sigma(z^i + tv^i)) \\ &= (\sigma'(z^i)v^i) \\ &= (\sigma(z^i)(1 - \sigma(z^i))v^i) \\ &= g(z) \odot (1 - g(z)) \odot v, \end{aligned}$$

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

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

2.2.2 The Hyperbolic Tangent Function

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

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

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

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

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

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

2.2.3 The Rectified Linear Unit Function

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

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

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.,

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

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

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

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

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

2.2.4 The Softmax Function

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

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

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

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

or rather in coordinates

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

where

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

2.3 Binary Classification - An Example

We return the network given by

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

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

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

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

$$\begin{aligned}\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) &= \frac{-1}{n} \sum_{j=1}^n (y_j \log(a^{[2]}_j) + (1 - y_j) \log(1 - a^{[2]}_j)) \\ &= \frac{-1}{n} (\langle y, \log(a^{[2]}) \rangle + \langle 1 - y, \log(1 - a^{[2]}) \rangle)\end{aligned}$$

Moreover, when using backpropagation, we see that

$$\begin{aligned}\delta^{[2]T}_j &= d(\mathbb{L}_{y_j})_{a^{[2]}} \cdot d(g^{[2]})_{z^{[2]}_j} \\ &= \left(-\frac{y_j}{a^{[2]}_j} + \frac{1 - y_j}{1 - a^{[2]}_j} \right) \cdot (a^{[2]}_j(1 - a^{[2]}_j)) \\ &= a^{[2]}_j - y_j,\end{aligned}$$

or rather

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

Similarly, we compute

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

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

```
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

```



```

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 @ delta2) * 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

```

```

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)

```

```

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:

$$\begin{array}{ccccccc}
 \underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} & \xrightarrow{\varphi^{[1]}} & \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} & \xrightarrow{g^{[1]}} & \underbrace{\begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 2}} & \xrightarrow{\varphi^{[2]}} & \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]s_2} \end{bmatrix}}_{\text{Layer 2}} & \xrightarrow{g^{[2]}} & \underbrace{\begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]s_2} \end{bmatrix}}_{\text{Layer 2}} & \xrightarrow{\varphi^{[3]}} \dots \\
 \dots & \xrightarrow{\varphi^{[L-1]}} & \underbrace{\begin{bmatrix} z^{[L-1]1} \\ \vdots \\ z^{[L-1]s_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} & \xrightarrow{g^{[L-1]}} & \underbrace{\begin{bmatrix} a^{[L-1]1} \\ \vdots \\ a^{[L-1]s_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} & \xrightarrow{\varphi^{[L]}} & \underbrace{\begin{bmatrix} z^{[L]1} \\ \vdots \\ z^{[L]s_L} \end{bmatrix}}_{\text{Layer } L} & \xrightarrow{g^{[L]}} & \underbrace{\begin{bmatrix} a^{[L]1} \\ \vdots \\ a^{[L]s_L} \end{bmatrix}}_{\text{Layer } L} & \Rightarrow \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{s_L} \end{bmatrix},
 \end{array}$$

where

$s_\ell :=$ the number of nodes in layer- ℓ ,

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

and

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

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 $\mathbb{L} : \mathbb{R}^{s_L} \times \mathbb{R}^{s_L} \rightarrow \mathbb{R}$ 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

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

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial W^{[\ell]}} &= \frac{1}{n} \delta^{[\ell]} a^{[\ell-1]T} \\ \frac{\partial \mathbb{J}}{\partial b^{[\ell]}} &= \frac{1}{n} \sum_{j=1}^n \delta^{[\ell]}_j\end{aligned}$$

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

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

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
```

```

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     """
91     w = {}
92     b = {}
93     for l in range(1, len(layers)):
94         w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
95         b[l] = np.zeros((layers[l], 1))
96     params = {'w' : w, 'b' : b}
97     return params
98
99 def forward_propagation(params, x, activators):
100     """
101     Parameters
102     -----
103     params : Dict[Dict]
104         params['w'][l] : array_like
105             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

```

```
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

Let $\mathbb{D} = \{(x_j, y_j) \in \mathbb{R}^m \times \mathbb{R}^K : 1 \leq j \leq N\}$ denote a dataset. Then we partition \mathbb{D} into three distinct sets

$$\mathbb{D} = \mathfrak{X} + \mathcal{D} + \mathcal{T},$$

where \mathfrak{X} is called our *training set*, \mathcal{D} is called our *development, or cross-validation set*, and \mathcal{T} is called our *test set*. We make this partition randomly, however, if $N = |\mathbb{D}| \leq 10^4$, we see a partition following the following ratios:

$$n_X := |\mathfrak{X}| \approx \frac{3}{5}N,$$

$$n_D := |\mathcal{D}| \approx \frac{1}{5}N,$$

and

$$n_T := |\mathcal{T}| \approx \frac{1}{5}N.$$

If however, we have a very large dataset (i.e., $N > 10^4$), then we assume a much smaller ratio of something similar to

$$\frac{n_X}{N} \approx 0.98, \quad \frac{n_D}{N} \approx 0.01, \quad \frac{n_T}{N} \approx 0.01.$$

In general, we use our training set \mathfrak{X} to train our parameters $W^{[\ell]}$ and $b^{[\ell]}$, we use our development set \mathcal{D} to tune our hyperparameters (i.e., learning rate, number of layers, number of nodes per layer, activation function, number of iterations to perform gradient descent, regularization parameters, etc), and we use our test set \mathcal{T} to evaluate the accuracy of our model. Since we're partitioning our dataset to better increase the accuracy of our model, we need to define an error function. To this end, define $\mathcal{E} : 2^{\mathbb{D}} \rightarrow [0, 1]$ by

$$\mathcal{E}(\mathcal{A}) = \frac{1}{|\mathcal{A}|} \sum_{(x,y) \in \mathcal{A}} \varepsilon(x, y),$$

where $\varepsilon : \mathbb{D} \rightarrow \{0, 1\}$ is defined by

$$\varepsilon(x, y) = \begin{cases} 1 & \text{if } y = \hat{y}(x) \\ 0 & \text{else.} \end{cases}$$

From our partition and error function we can make several claims of the fitting of our model to our data. Indeed, let $\epsilon > 0$ be a small percentage (with exact value depending on specific examples), then:

- If $\mathcal{E}(\mathcal{X}) < \epsilon$ and $\mathcal{E}(\mathcal{X}) < \mathcal{E}(\mathcal{D}) < \sim 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.
- If $\mathcal{E}(\mathcal{X}) \approx \mathcal{E}(\mathcal{D}) > \sim 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.
- If $10\epsilon \sim < \mathcal{E}(\mathcal{X}) \ll \mathcal{E}(\mathcal{D})$, then we say our model has both high bias (since it doesn't fit our training data well) and high variance (because the model fits the training data better than the development data).
- If $\mathcal{E}(\mathcal{X}), \mathcal{E}(\mathcal{D}) < \epsilon$, then we say the model has both low bias and low variance.

Remark 4.1. *The interpretations of our error percentage is based on two crucial assumptions:*

- \mathcal{D} and \mathcal{T} come from samplings with the same distribution of outputs (i.e., if we're determining whether a collection of images contain a cat, we should never have that \mathcal{D} is mostly cat pictures, and \mathcal{T} is mostly non-cat pictures).
- The optimal error for the model is approximately 0%. That is, if a human were looking at the data, they could determine the correct response with negligible error. This is sometimes called the Bayes error.

If either of these assumptions fail to hold, other methods of analysis may be required to obtain meaningful insights for the performance of our model.

A methodology for using errors could be as follows

1. Check $\mathcal{E}(\mathcal{X})$ for high bias.
 - a. If “Yes”, then we can try a bigger network, we can train longer, or we can change the neural network architecture. Then we return to (1.).
 - b. If “No”, then we move to (2.).
2. Check $\mathcal{E}(\mathcal{D})$ for high variance.
 - a. If “Yes”, then we can try to get more data, try regularization, or try changing the neural network architecture. Then we return to (1.).
 - b. If “No”, then we're done.

4.0.1 Python Implementation

To implement a partitioning we could do something like the following:

```
1 import numpy as np
2 from sklearn.utils import shuffle
3
4 def partition_data(x, y, train_ratio):
5     """
6     Parameters
7     -----
8     x : array_like
9         x.shape = (m, N)
10    y : array_like
11        y.shape = (k, N)
12    train_ratio : float
13        0<=train_ratio<=1
14
15    Returns
16    -----
17    train : Tuple[array_like]
18    dev : Tuple[array_like]
19    test : Tuple[array_like]
20    """
21    ## Shuffle the data
22    x, y = shuffle(x.T, y.T) #
23    x = x.T
24    y = y.T
25
26    ## Get the size of partitions
27    N = x.shape[1]
28    N_train = int(train_ratio * N)
29    N_mid = (N - N_train) // 2
30
31    ## Create partitions
32    train = (x[:, :N_train], y[:, :N_train])
33    dev = (x[:, N_train:N_train+N_mid], y[:, N_train:N_train+N_mid])
34    test = (x[:, N_train+N_mid:], y[:, N_train+N_mid:])
35
36    assert(x.all() == np.concatenate([train[0], dev[0], test[0]], axis=1).all())
37    assert(y.all() == np.concatenate([train[1], dev[1], test[1]], axis=1).all())
38
39    return train, dev, test
```

5 Regularization

Suppose we're training an L -layer neural network with dataset $\{(x_j, y_j)\} \subset \mathbb{R}^{s_0} \times \mathbb{R}^{s_L}$ with N examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{s_L} \times \mathbb{R}^{s_L} \rightarrow \mathbb{R}$, then we have our cost function \mathbb{J} defined on our one-parameter families of parameters W and b given by

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

If our model suffers from overfitting the training set, it's reasonable to impose constraints on the parameters W and/or b . That is, define the function

$$R(W) = \frac{\lambda}{2N} \sum_{\ell=1}^L \|W^{[\ell]}\|_F^2,$$

for some $\lambda > 0$, where $\|\cdot\|_F$ represents the Frobenius norm on matrices, and we define the *regularized cost function* \mathbb{J}^R given by

$$\begin{aligned} \mathbb{J}^R(W, b) &= \mathbb{J}(W, b) + R(W) \\ &= \frac{1}{N} \sum_{j=1}^N \mathbb{L}(\hat{y}_j, y_j) + \frac{\lambda}{2N} \sum_{\ell=1}^L \|W^{[\ell]}\|_F^2. \end{aligned}$$

Adding such an $R(W)$ to our cost function is known as L^2 -*regularization*. We note that by linearity we have the following equalities amongst gradients:

$$\frac{\partial \mathbb{J}^R}{\partial b^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial b^{[\ell]}}$$

and

$$\frac{\partial \mathbb{J}^R}{\partial W^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial W^{[\ell]}} + \frac{\lambda}{N} W^{[\ell]}.$$

The idea behind regularization is that we're now minimizing

$$\min_{W, b} \mathbb{J}^R(W, b) = \min_{W, b} \{\mathbb{J}(W, b) + R(W)\},$$

and so for suitably chosen $\lambda > 0$, it forces $\|W^{[\ell]}\|_F$ to be small, along with minimizing the cost \mathbb{J} . This balancing-act of minimizing the two functions simultaneously helps with overfitting the data.

A typical usage of regularization would be similar to the following outline:

- i. Partition our dataset $\mathbb{D} = \mathfrak{X} \cup \mathcal{D} \cup \mathcal{T}$.
- ii. Give a set Λ of potential regularization parameters.
- iii. For each $\lambda \in \Lambda$, we first train on \mathfrak{X} , that is, we obtain

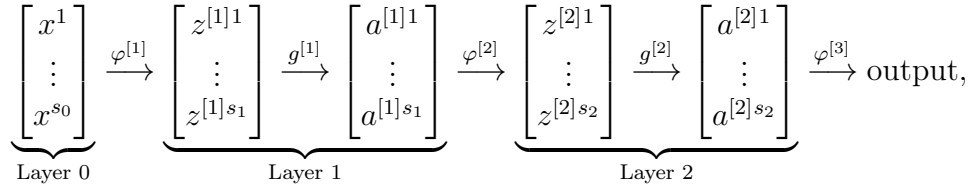
$$\begin{aligned} (W, b) &= \arg \min_{W, b} \mathbb{J}^R(W, b) \\ &= \arg \min_{W, b} \left\{ \frac{1}{n_X} \sum_{(x, y) \in \mathfrak{X}} \mathbb{L}(\hat{y}, y) + \frac{\lambda}{2n_X} \sum_{\ell=1}^L \|W^{[\ell]}\|_F^2 \right\} \end{aligned}$$

which dependent on λ .

- iv. Then using the aforementioned $(W, b) = (W, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathfrak{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.
- v. After finding $\mathcal{E}_\lambda(\mathfrak{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$ for each $\lambda \in \Lambda$, we choose our desired λ and hence our desired parameters W and b .
- vi. We evaluate our model on \mathcal{T} to determine the overall accuracy.

5.1 (Inverted) Dropout Regularization

For illustrative purposes, suppose we have a 3-layer neural network of the following form:



Let Q_0, Q_1, Q_2 denote the collection of all nodes in Layers 0, 1, 2, respectively. Let $p_0, p_1, p_2 \in [0, 1]$, and define a probability distribution \mathbb{P}_ℓ on Q_ℓ by

$$\mathbb{P}_\ell(q = 1) = p_\ell, \quad \mathbb{P}_\ell(q = 0) = 1 - p_\ell,$$

where $q = 1$ represents the node existing in layer- ℓ , and $q = 0$ represents the dropping of the node from layer- ℓ . That is we're effectively reducing the number of nodes throughout the network, thus simplifying the network and reducing the amount of influence of any single feature or node on the entire model. That is, we would implement a methodology similar to the following:

- i. For each layer ℓ and each training example x_j define the “dropout vector” $D^{[\ell]}_j$ by

$$D^{[\ell]}_j = \begin{bmatrix} d_j^1 \\ \vdots \\ d_j^{s_\ell} \end{bmatrix},$$

where

$$d_j^i = \begin{cases} 1 & \text{if } \mathbb{P}(q^i) \leq p_\ell \\ 0 & \text{if } \mathbb{P}(q^i) > p_\ell \end{cases}.$$

- ii. During forward propagation, we redefine

$$a^{[\ell]} \mapsto \frac{a^{[\ell]} \odot D^{[\ell]}}{p_\ell}.$$

- iii. During backward propagation, we define

$$\delta^{[\ell]} \mapsto \frac{\delta^{[\ell]} \odot D^{[\ell]}}{p_\ell}.$$

- iv. Then perform gradient descent, etc with these new values.

5.1.1 Python Implementation

We see here the use of inverted dropout regularization in a general neural network.

```

1 import copy
2
3 import numpy as np
4
5 import utils
6
7 def dropout_matrices(layers, num_examples, keep_prob):
8     """
9     Parameters
10    -----
11    layers : List[int]
12        layers[1] = number of nodes in layer 1
13    num_examples : int
14        The number of training examples
15    keep_prob : List[float]
```

```

16         keep_prob[l] = The probabilitly of keeping a node in layer l
17
18     Returns
19     -----
20     D : Dict[array_like]
21         D[l].shape = (layers[l], num_ex)
22         D[l] = a Boolean array
23     """
24     np.random.seed(1)
25     L = len(layers)
26     D = {}
27     for l in range(L - 1):
28         D[l] = np.random.rand((layers[l], num_examples))
29         D[l] = D[l] < keep_prob[l]
30         assert(D[l].shape == (layers[l], num_examples))
31     return D
32
33 def linear_activation_forward(a_prev, w, b, activator):
34     """
35     Parameters
36     -----
37     a_prev : array_like
38         a_prev.shape = (layers[l], n)
39     w : array_like
40         w.shape = (layers[l+1], layers[l])
41     b : array_like
42         b.shape = (layers[l+1], 1)
43     activator : str
44         activator = 'relu', 'sigmoid', 'tanh', 'softmax'
45
46     Returns
47     -----
48     z : array_like
49         z.shape = (layer_dims[l+1], n)
50     a : array_like
51         a.shape = (layer_dims[l+1], n)
52     """
53     z = w @ a_prev + b
54     if activator == 'relu':
55         a, _ = utils.relu(z)
56     elif activator == 'sigmoid':
57         a, _ = utils.sigmoid(z)
58     else:
59         print("Activation_function_doesn't_match_ReLu_or_sigmoid.")
60     return z, a
61
62 def forward_propagation(params, D, keep_prob, x):

```

```

63     """
64     Parameters
65     -----
66     params : Dict[Dict]
67         params['w'][l] : array_like
68             wl.shape = (layers[l], layers[l-1])
69         params['b'][l] : array_like
70             bl.shape = (layers[l], 1)
71     D : Dict[array_like]
72         D[l].shape = (layer_dims[l], num_ex)
73         D[l] = a Boolean array
74     keep_prob : List[float]
75         keep_prob[l] = The probability of keeping a node in layer l
76     x : array_like
77         x.shape = (layers[0] n)
78
79     Returns
80     -----
81     cache : Dict[Dict]
82         cache['z'][l] : array_like
83             zl.shape = (layers[l], n)
84         cache['a'][l] : array_like
85             al.shape = (layers[l], n)
86     """
87     # Retrieve parameters
88     w = params['w']
89     b = params['b']
90     L = len(w) + 1 # Number of layers including input layer
91     n = x.shape[1]
92
93     # Set empty caches
94     a = {}
95     z = {}
96     # Dropout on layer 0
97     a[0] = x
98     a[0] = a[0] @ D[0]
99     a[0] /= keep_prob[0]
100    # Loop through hidden layers
101    for l in range(1, L):
102        zl, al = linear_activation_forward(a[l - 1], w[l], b[l], 'relu')
103        al = al @ D[l]
104        al /= keep_prob[l]
105        z[l] = zl
106        a[l] = al
107
108    # Output layer
109    z[L], a[L] = linear_activation_forward(a[L - 1], w[L], b[L], 'sigmoid')

```

```

110
111     cache = {'z' : z, 'a' : a}
112     return cache
113
114 def linear_activation_backward(delta_next, z, w, activator):
115     """
116     Parameters
117     -----
118     delta_next : array_like
119         delta_next.shape = (layers[l+1], n)
120     z : array_like
121         z.shape = (layers[l+1], n)
122     w : array_like
123         w.shape = (layers[l+1], layers[l])
124     activator : str
125         activator = 'relu', 'sigmoid', 'tanh', 'softmax'
126
127     Returns
128     -----
129     delta : array_like
130         delta.shape = (layers[l])
131     """
132     n = delta_next.shape[1]
133
134     if activator == 'relu':
135         _, dg = relu(z)
136     elif activator == 'sigmoid':
137         _, dg = sigmoid(z)
138     else:
139         print("Activation_function_doesn't_match_ReLu_or_sigmoid.")
140
141     da = w.T @ delta_next
142     assert(da.shape == (w.shape[0], n))
143     delta = da * dg
144     assert(delta.shape == (w.shape[0], n))
145     return delta
146
147 def backward_propagation(params, cache, D, keep_prob, x, y):
148     """
149     Parameters
150     -----
151     params : Dict
152         params['w'][l] : array_like
153             w[l].shape = (layers[l], layers[l-1])
154         params['b'][l] : array_like
155             b[l].shape = (layers[l], 1)
156     cache : Dict

```



```

157         cache['a'][l] : array_like
158         a[l].shape = (layers[l], n)
159         cache['z'][l] : array_like
160         z[l].shape = (layers[l], n)
161     D : Dict[array_like]
162         D[l].shape = (layer[l], num_ex)
163         D[l] = a Boolean array
164     keep_prob : List[float]
165         keep_prob[l] = The probability of keeping a node in layer l
166     x : array_like
167         x.shape = (layers[0], n)
168     y : array_like
169         y.shape = (layers[-1], n)
170     Returns
171     -----
172     grads : Dict[Dict]
173         grads['dw'][l] : array_like
174             dw[l].shape = w[l].shape
175         grads['db'][l] : array_like
176             db[l].shape = b[l].shape
177     """
178     ## Retrieve parameters
179     a = cache['a']
180     z = cache['z']
181     w = params['w']
182     n = x.shape[1]
183     L = len(z)
184
185     ## Compute deltas
186     delta = {}
187     delta[L] = a[L] - y
188     for l in reversed(range(1, L)):
189         delta = linear_activation_backward(delta[l + 1], z[l], w[l], 'relu')
190         delta = delta @ D[l]
191         delta /= keep_prob[l]
192
193     ## Compute gradients
194     dw = {}
195     db = {}
196
197     for l in range(1, L + 1):
198         db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
199         assert(db[l].shape == (w[l].shape[0], 1))
200         dw[l] = (1 / n) * delta[l] * a[l - 1].T
201         assert(dw[l].shape == w[l].shape)
202     grads = {'dw' : dw, 'db' : db}
203     return grads

```

```

204
205 def model(x, y,
206           hidden_sizes,
207           keep_prob,
208           activators,
209           num_iters=2500,
210           learning_rate=0.1,
211           print_cost=False):
212     """
213     Parameters
214     -----
215     Parameters
216     -----
217     x : array_like
218         x.shape = (layers[0], n)
219     y : array_like
220         y.shape = (layers[-1], n)
221     hidden_sizes : List[int]
222         The number nodes layer l = hidden_sizes[l-1]
223     activators : List[function]
224         activators[l] = activation function of layer l+1
225     num_iters : int
226         Number of iterations with which our model performs gradient descent
227     learning_rate : float
228         The learning rate for gradient descent
229     print_cost : Boolean
230         If True, print the cost every 1000 iterations
231
232     Returns
233     -----
234     params : Dict[Dict]
235         params['w'][l] : array_like
236             w[l].shape = (layers[l], layers[l-1])
237         params['b'][l] : array_like
238             b[l].shape = (layers[l], 1)
239     cost : float
240         The final cost value for the optimized parameters returned
241     """
242     n, layers = dim_retrieval(x, y, hidden_sizes)
243     params = initialize_parameters(layers)
244     for i in range(num_iters):
245         D = dropout_matrices(layers, n, keep_prob)
246         cache = forward_propagation(params, D, keep_prob, x)
247         cost = utils.compute_cost(cache, y)
248         grads = backward_propagation(params, cache, D, keep_prob, x, y)
249         params = utils.update_parameters(params, grads, learning_rate)
250

```

```
251         if print_cost and i % 1000 == 0:
252             print(f'Cost_after_iteration_{i}:_{cost}')
253
254     return params, cost
```

5.2 Data Augmentation

This section requires work.

There are few other regularization techniques. One of the simplest techniques is data augmentation, i.e., transforming data you currently have into related but different example to gather a larger dataset (e.g., flipping or distorting images to obtain other relevant images).

5.3 Early Stopping

This section requires work.

Another technique is stop the training early (fewer iterations) before the model develops higher variance.

6 Gradients and Numerical Remarks

This section requires work

We first remark, that by our use of gradient descent, there are few outlier cases which may occur. Namely our gradients may explode or vanish. One way to attempt to fix such a situation to impose a normalization on our weights depending on our activation functions.

- If $g^{[\ell]} = \text{ReLU}$, then we wish to impose the requirement that

$$\mathbb{E}[(W^{[\ell]2})] = \frac{1}{s_{\ell-1}}.$$

6.1 Numerical Gradient Checking

Suppose $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is a smooth function. Then, we recall the definition of the partial derivative

$$\begin{aligned} \frac{\partial f}{\partial x^j} &= \lim_{h \rightarrow 0} \frac{f(x + he_j) - f(x)}{h} \\ &= \lim_{\epsilon \rightarrow 0^+} \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon}, \end{aligned}$$

and so for sufficiently small $\epsilon > 0$, we have the approximation

$$\frac{\partial f}{\partial x^j} \approx \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon}.$$

Define the approximation function $F : \mathbb{R}^n \times (0, 1) \rightarrow \mathbb{R}^n$ by

$$F(x, \epsilon) = \frac{1}{2\epsilon} \begin{bmatrix} f(x + \epsilon e_1) - f(x - \epsilon e_1) \\ \vdots \\ f(x + \epsilon e_n) - f(x - \epsilon e_n) \end{bmatrix}.$$

Then we may check that our gradient computation $\nabla f(x)$ is correct by checking that

$$\frac{\|F(x, \epsilon) - \nabla f(x)\|_2}{\|F(x, \epsilon)\|_2 + \|\nabla f(x)\|_2} \approx 0.$$

6.1.1 Python Implementation

```
1 ## f(x) = x_1*x_2*...*x_n
2 def fctn(x):
3     n = x.shape[0]
4     f = np.prod(x)
5     grad = np.zeros((n, 1))
6     for i in range(n):
7         omit = 1 - np.eye(1, n, i).T
8         omit = np.array(omit, dtype=bool)
9         grad[i, 0] = np.prod(x, where=omit)
10    return f, grad
11
12 def gradient_check(grad, f, x, epsilon=1e-3):
13     """
14     Parameters
15     -----
16     grad : array_like
17         grad.shape= (n, 1)
18     f : function
19         The function to check.
20     x : array_like
21         x.shape = (n, 1)
22     epsilon : float
23         Default 0.001
24     Returns
25     error : float
26     -----
27     """
28     n = x.shape[0]
29     f_diffs = []
30     for i in range(n):
31         e = np.eye(1, n, i).T
32         x_plus = x + epsilon * e
33         x_minus = x - epsilon * e
34         f_plus, _ = f(x_plus)
35         f_minus, _ = f(x_minus)
36         f_diffs.append(f_plus - f_minus)
37     f_diffs = np.array(f_diffs).reshape(n, 1)
38     f_diffs = f_diffs / (2 * epsilon)
39
40     error = (np.linalg.norm(f_diffs - grad)
41             / (np.linalg.norm(f_diffs) + np.linalg.norm(grad)))
42     return error
```