

Neural Networks

Matt R

March 18, 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	28
3.1	Backpropagation	28
3.1.1	Vectorization in Python	29
II	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization	37

4	Training, Development and Test Sets	38
4.0.1	Python Implementation	40
5	Regularization	41
5.0.1	Python Implementation	42
5.1	(Inverted) Dropout Regularization	46
5.1.1	Python Implementation	47
5.2	Data Augmentation	52
5.3	Early Stopping	52
6	Gradients and Numerical Remarks	53
6.1	Numerical Gradient Checking	53
6.1.1	Python Implementation	54
7	Gradient Descent	55
7.1	Weighted Averages	57
7.2	Gradient Descent with Momentum	59
7.3	Root Mean Squared Propagation (RMSProp)	61
7.4	Adaptive Moment Estimation: The Adam Algorithm	62
7.5	Learning Rate Decay	64
7.5.1	Python Implementation	64
8	Tuning Hyper-Parameters	72
8.0.1	Python Implementation	73
9	Batch Normalization	74

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, for function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ 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,
134          y_train,
135          x_test,
136          y_test,
137          learning_rate=0.001,
138          num_iters=2000, accuracy=False):
139     """
140     Parameters:
141     -----
142     x_train, y_train, x_test, y_test : array_like
143         x_train.shape = (m, n_train)
144         y_train.shape = (1, n_train)
145         x_test.shape = (m, n_test)
146         y_test.shape = (1, n_test)
147     learning_rate : float
148         The learning rate for gradient descent
149     num_iters : int
150         The number of times we wish to perform gradient descent
151     accuracy : Boolean
152         Use True to print the accuracy of the model
153
154     Returns:
155     d : Dict
156         d['costs'] : array_like
157         The costs evaluated every 100 iterations

```

```

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

```

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

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

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

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

Definition 2.1. Suppose $g : \mathbb{R} \rightarrow \mathbb{R}$ 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

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1}_{\mathbb{R}^{s_2 \times s_1}} \times \mathbb{1}_{\mathbb{R}^{s_2}} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

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

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

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

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

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 import activators
6 from activators import ACTIVATORS
7
8 # Preliminary functions for our model
9 def dim_retrieval(x, y, hidden_sizes):
10     """
11     Parameters
12     -----
13     x : array_like
14         x.shape = (layers[0], n)
15     y : array_like
16         y.shape = (layers[L], n)
17     hidden_sizes : List[int]
18         hidden_sizes[i-1] = The number nodes layer i
19     Returns
20     -----
21     n : int
22         The number of training examples
23     layers : List
24         layer[1] = # nodes in layer 1
25
26     """
27     m, n = x.shape
28     assert(y.shape[1] == n)
29     K = y.shape[0]
30     layers = [m]
31     layers.extend(hidden_sizes)
32     layers.append(K)
33
34     return n, layers
35
36 ## Initialize parameters using the size of each layer
37 def initialize_parameters_random(layers):
38     """
39     Parameters
40     -----
41     layers : List[int]
42         layers[1] = # nodes in layer 1
43     Returns
44     -----
45     params : Dict[Dict]
```

```

46         w[l] : array_like
47         dwl.shape = (layers[l], layers[l-1])
48         b[l] : array_like
49         dbl.shape = (layers[l], 1)
50     """
51     w = {}
52     b = {}
53     for l in range(1, len(layers)):
54         w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
55         b[l] = np.zeros((layers[l], 1))
56     params = {'w' : w, 'b' : b}
57     return params
58
59 def forward_propagation(x, params):
60     """
61     Parameters
62     -----
63     x : array_like
64         x.shape = (m_x, n)
65     params : Dict[Dict]
66         w[l] : array_like
67             w[l].shape = (layers[l], layers[l-1])
68         b[l] : array_like
69             b[l].shape = (layers[l], 1)
70     Returns
71     -----
72     a2 : array_like
73         a2.shape = (m_y, n)
74     cache : Dict
75         cache['z1'] : array_like
76             z1.shape = (m_h, n)
77         cache['a1'] : array_like
78             a1.shape = (m_h, n)
79         cache['z2'] : array_like
80             z2.shape = (m_y, n)
81         cache['a2'] = a2
82     """
83
84     # Retrieve parameters
85     w = params['w']
86     b = params['b']
87     w1 = w[1]
88     b1 = b[1]
89     w2 = w[2]
90     b2 = b[2]
91
92     # Auxiliary computations

```

```

93     z1 = w1 @ x + b1
94     a1, _1 = activators.tanh(z1)
95     z2 = w2 @ a1 + b2
96     a2, _2 = activators.sigmoid(z2)
97
98     assert(a1.shape == (w1.shape[0], x.shape[1]))
99     assert(a2.shape == (w2.shape[0], a1.shape[1]))
100
101     cache = {'z1' : z1,
102             'a1' : a1,
103             'z2' : z2,
104             'a2' : a2}
105
106     return a2, cache
107
108 def compute_cost(a2, y):
109     """
110     Parameters
111     -----
112     a2 : array_like
113         a2.shape = (m_y, n)
114     y : array_like
115         y.shape = (m_y, n)
116     Returns
117     -----
118     cost : float
119         The cost evaluated at y and a2
120     """
121     n = y.shape[1]
122     cost = (-1 / n) * (np.sum(y * np.log(a2)) + np.sum((1 - y) * np.log(1 - a2)))
123     cost = float(np.squeeze(cost)) # Makes sure we return a float
124
125     return cost
126
127 def backward_propagation(params, cache, x, y):
128     """
129     Parameters
130     -----
131     params : Dict[Dict]
132         w[1] : array_like
133             dw1.shape = (layers[1], layers[1-1])
134         b[1] : array_like
135             db1.shape = (layers[1], 1)
136     cache : Dict
137         cache['z1'] : array_like
138             z1.shape = (m_h, n)
139         cache['a1'] : array_like

```



```

140         a1.shape = (m_h, n)
141         cache['z2'] : array_like
142         z2.shape = (m_y, n)
143         cache['a2'] = a2
144     x : array_like
145         x.shape = (m_x, n)
146     y : array_like
147         y.shape = (m_y, n)
148     Returns
149     -----
150     grads : Dict
151         grads['dw2'] : array_like
152             dw2.shape = (m_y, m_h)
153         grads['db2'] : array_like
154             db2.shape = (m_y, 1)
155         grads['dw1'] : array_like
156             dw1.shape = (m_h, m_x)
157         grads['db1'] : array_like
158             db1.shape = (m_h, 1)
159     """
160     # Retrieve parameters
161     w = params['w']
162     w1 = w[1]
163     w2 = w[2]
164
165     # Set dimensional constants
166     m_x, n = x.shape
167     m_y, m_h = w2.shape
168
169     # Retrieve node outputs
170     a1 = cache['a1']
171     a2 = cache['a2']
172
173     # Auxiliary Computations
174     delta2 = a2 - y
175     assert(delta2.shape == (m_y, n))
176     d_tanh = 1 - (a1 * a1)
177     assert(d_tanh.shape == (m_h, n))
178     delta1 = (w2.T @ delta2) * d_tanh
179     assert(delta1.shape == (m_h, n))
180
181     # Gradient computations
182     dw = {}
183     db = {}
184     dw[2] = (1 / n) * delta2 @ a1.T
185     db[2] = (1 / n) * np.sum(delta2, axis=1, keepdims=True)
186     dw[1] = (1 / n) * delta1 @ x.T

```

```

187     db[1] = (1 / n) * np.sum(delta1, axis=1, keepdims=True)
188
189     # Combine and return dict
190     grads = {'dw' : dw, 'db' : db}
191     return grads
192
193 def update_parameters(params, grads, learning_rate=1.2):
194     """
195     Parameters
196     -----
197     params : Dict
198         params['w2'] : array_like
199             w2.shape = (m_y, m_h)
200         params['b2'] : array_like
201             b2.shape = (m_y, 1)
202         params['w1'] : array_like
203             w1.shape = (m_h, m_x)
204         params['b1'] : array_like
205             b1.shape = (m_h, 1)
206     grads : Dict
207         grads['dw2'] : array_like
208             dw2.shape = (m_y, m_h)
209         grads['db2'] : array_like
210             db2.shape = (m_y, 1)
211         grads['dw1'] : array_like
212             dw1.shape = (m_h, m_x)
213         grads['db1'] : array_like
214             db1.shape = (m_h, 1)
215     learning_rate : float
216         Default = 1.2
217     Returns
218     -----
219     params : Dict
220         params['w2'] : array_like
221             w2.shape = (m_y, m_h)
222         params['b2'] : array_like
223             b2.shape = (m_y, 1)
224         params['w1'] : array_like
225             w1.shape = (m_h, m_x)
226         params['b1'] : array_like
227             b1.shape = (m_h, 1)
228     """
229     # Retrieve parameters
230     w = copy.deepcopy(params['w'])
231     b = params['b']
232
233     # Retrieve gradients

```

```

234     dw = grads['dw']
235     db = grads['db']
236
237     # Perform update
238     w[2] = w[2] - learning_rate * dw[2]
239     b[2] = b[2] - learning_rate * db[2]
240     w[1] = w[1] - learning_rate * dw[1]
241     b[1] = b[1] - learning_rate * db[1]
242
243     # Combine and return dict
244     params = {'w' : w, 'b' : b}
245     return params
246
247
248 # The main neural network training model
249 def model(x, y, hidden_sizes, num_iters=10000, print_cost=False):
250     """
251     Parameters
252     -----
253     x : array_like
254         x.shape = (m_x, n)
255     y : array_like
256         y.shape = (m_y, n)
257     hidden_sizes : int
258         Number of nodes in the single hidden layer
259     num_iters : int
260         Number of iterations with which our model performs gradient descent
261     print_cost : Boolean
262         If True, print the cost every 1000 iterations
263     Returns
264     -----
265     params : Dict[Dict[array_like]]
266         params['w'][2] : array_like
267             w[2].shape = (m_y, m_h)
268         params['b'][2] : array_like
269             b[2].shape = (m_y, 1)
270         params['w'][1] : array_like
271             w[1].shape = (m_h, m_x)
272         params['b'][1] : array_like
273             b[1].shape = (m_h, 1)
274     """
275     # Set dimensional constants
276     n, layers = dim_retrieval(x, y, hidden_sizes)
277     # initialize parameters
278     params = initialize_parameters_random(layers)
279
280     # main loop for gradient descent

```

```

281     for i in range(num_iters):
282         a2, cache = forward_propagation(x, params)
283         cost = compute_cost(a2, y)
284         grads = backward_propagation(params, cache, x, y)
285         params = update_parameters(params, grads)
286
287         if print_cost and i % 1000 == 0:
288             print(f'Cost_after_iteration_{i}:_{cost}')
289
290     return params
291
292 # Using our model to obtain predictions
293 def predict(params, x):
294     """
295     Parameters
296     -----
297     params : Dict
298         params['w2'] : array_like
299             w2.shape = (m_y, m_h)
300         params['b2'] : array_like
301             b2.shape = (m_y, 1)
302         params['w1'] : array_like
303             w1.shape = (m_h, m_x)
304         params['b1'] : array_like
305             b1.shape = (m_h, 1)
306     x : array_like
307         x.shape = (m_x, n)
308
309     Returns
310     -----
311     predictions : array_like
312         predictions.shape = (m_y, n)
313     """
314     a2, _ = forward_propagation(x, params)
315     predictions = np.zeros(a2.shape)
316     predictions[~(a2 < 0.5)] = 1
317
318     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 import utils
6 import activators
7 from activators import ACTIVATORS
8
9
10 ## Auxiliary functions for model composition
```

```

11
12
13 def initialize_parameters(layers):
14     """
15     Parameters
16     -----
17     layers : List[int]
18         layers[l] = # nodes in layer l
19     Returns
20     -----
21     params : Dict[Dict]
22         w[l] : array_like
23             dwl.shape = (layers[l], layers[l-1])
24         b[l] : array_like
25             dbl.shape = (layers[l], 1)
26     """
27     w = {}
28     b = {}
29     for l in range(1, len(layers)):
30         w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
31         b[l] = np.zeros((layers[l], 1))
32     params = {'w' : w, 'b' : b}
33     return params
34
35 ## Compute activation unit
36 def linear_activation_forward(a_prev, w, b, activator):
37     """
38     Parameters
39     -----
40     a_prev : array_like
41         a_prev.shape = (layers[l], n)
42     w : array_like
43         w.shape = (layers[l+1], layers[l])
44     b : array_like
45         b.shape = (layers[l+1], 1)
46     activator : str
47         activator = 'relu', 'sigmoid', or 'tanh'
48
49     Returns
50     -----
51     z : array_like
52         z.shape = (layer_dims[l+1], n)
53     a : array_like
54         a.shape = (layer_dims[l+1], n)
55     """
56     assert activator in ACTIVATORS, f'{activator}_is_not_a_valid_activator.'
57

```

```

58     z = w @ a_prev + b
59     if activator == 'relu':
60         a, _ = activators.relu(z)
61     elif activator == 'sigmoid':
62         a, _ = activators.sigmoid(z)
63     elif activator == 'tanh':
64         a, _ = activators.tanh(z)
65
66     assert(z.shape == a.shape)
67     return z, a
68
69 def forward_propagation(x, params, activators):
70     """
71     Parameters
72     -----
73     x : array_like
74         x.shape = (layers[0] n)
75     params : Dict[Dict]
76         params['w'][l] : array_like
77             w_l.shape = (layers[l], layers[l-1])
78         params['b'][l] : array_like
79             b_l.shape = (layers[l], 1)
80     activators : List[str]
81         activators[l] = activation function of layer l+1
82     Returns
83     -----
84     cache : Dict[Dict]
85         cache['z'][l] : array_like
86             z[l].shape = (layers[l], n)
87         cache['a'][l] : array_like
88             a[l].shape = (layers[l], n)
89     """
90     # Retrieve parameters
91     w = params['w']
92     b = params['b']
93     L = len(w) # Number of layers excluding output layer
94     n = x.shape[1]
95     # Set empty caches
96     a = {}
97     z = {}
98     # Initialize a
99     a[0] = x
100    for l in range(1, L + 1):
101        z[l], a[l] = linear_activation_forward(a[l - 1], w[l], b[l], activators[l -
102
103    cache = {'a' : a, 'z' : z}
104    return cache

```



```

105
106 # Compute the cost
107 def compute_cost(y, cache):
108     """
109     Parameters
110     -----
111     y : array_like
112         y.shape = (layers[-1], n)
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     Returns
120     -----
121     cost : float
122         The cost evaluated at y and aL
123     """
124     ## Retrieve parameters
125     n = y.shape[1]
126     a = cache['a']
127     L = len(a)
128     aL = a[L - 1]
129
130     cost = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
131     cost = float(np.squeeze(cost))
132
133     return cost
134
135 def linear_activation_backward(delta_next, z, w, activator):
136     """
137     Parameters
138     -----
139     delta_next : array_like
140         delta_next.shape = (layers[l+1], n)
141     z : array_like
142         z.shape = (layers[l+1], n)
143     w : array_like
144         w.shape = (layers[l+1], layers[l])
145     activator : str
146         activator = 'relu', 'sigmoid', or 'tanh'
147
148     Returns
149     -----
150     delta : array_like
151         delta.shape = (layers[l], n)

```

```

152     """
153     assert activator in ACTIVATORS, f'{activator}_is_not_a_valid_activator.'
154
155     n = delta_next.shape[1]
156
157     if activator == 'relu':
158         _, dg = activators.relu(z)
159     elif activator == 'sigmoid':
160         _, dg = activators.sigmoid(z)
161     elif activator == 'tanh':
162         _, dg = activators.tanh(z)
163
164     da = w.T @ delta_next
165     assert(da.shape == (w.shape[1], n))
166     delta = da * dg
167     assert(delta.shape == (w.shape[1], n))
168     return delta
169
170 def backward_propagation(x, y, params, cache, activators):
171     """
172     Parameters
173     -----
174     x : array_like
175         x.shape = (layers[0], n)
176     y : array_like
177         y.shape = (layers[-1], n)
178     params : Dict[Dict[array_like]]
179         params['w'][1] : array_like
180             w[1].shape = (layers[1], layers[1-1])
181         params['b'][1] : array_like
182             b[1].shape = (layers[1], 1)
183     cache : Dict[Dict[array_like]]
184         cache['a'][1] : array_like
185             a[1].shape = (layers[1], n)
186         cache['z'][1] : array_like
187             z[1].shape = (layers[1], n)
188     activators : List[str]
189         activators[1] = activation function of layer 1+1
190     Returns
191     -----
192     grads : Dict[Dict]
193         grads['dw'][1] : array_like
194             dw[1].shape = w[1].shape
195         grads['db'][1] : array_like
196             db[1].shape = b[1].shape
197     """
198     ## Retrieve parameters

```

```

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

```

```

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

```

```

293     """
294     ## Set dimensions and Initialize parameters
295     n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
296     params = utils.initialize_parameters_random(layers)
297
298     ## main loop
299     for i in range(num_iters):
300         cache = forward_propagation(x, params, activators)
301         cost = compute_cost(cache, y)
302         grads = backward_propagation(x, y, params, cache, activators)
303         params = update_parameters(params, grads, 0.1)
304
305         if print_cost and i % 1000 == 0:
306             print(f'Cost_after_iteration_{i}:_{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

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

where \mathbb{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 being divided accordingly to the following ratios:

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

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

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 \mathbb{X} to train our parameters $W^{[\ell]}$ and $b^{[\ell]}$, we use our development set \mathcal{D} to tune our hyper-parameters (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 $\mathcal{E}(\mathbb{X}) < \epsilon$ and $\mathcal{E}(\mathbb{X}) < \mathcal{E}(\mathcal{D}) \lesssim 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.
- If $\mathcal{E}(\mathbb{X}) \approx \mathcal{E}(\mathcal{D}) \gtrsim 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.
- If $10\epsilon \lesssim \mathcal{E}(\mathbb{X}) \ll \mathcal{E}(\mathcal{D})$, 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 $\mathcal{E}(\mathbb{X}), \mathcal{E}(\mathcal{D}) < \epsilon$, 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 $\mathcal{E}(\mathbb{X})$ 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 $\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}$.
- ii. Give a set Λ of potential regularization parameters.
- iii. For each $\lambda \in \Lambda$, we first train on \mathbb{X} , that is, we obtain

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

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

which dependent on λ .

- iv. Then using the aforementioned $(W, b) = (W, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.
- v. After finding $\mathcal{E}_\lambda(\mathbb{X})$ 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.

5.0.1 Python Implementation

```

1 import numpy as np
2
3 import utils
4 import activators
5
6 def forward_propagation(x, params, activators):
7     """
8     Parameters
9     -----
10    x : array_like
11        x.shape = (layers[0] n)
12    params : Dict[Dict]
13        params['w'][1] : array_like
14            wl.shape = (layers[1], layers[1-1])
15        params['b'][1] : array_like
16            bl.shape = (layers[1], 1)
17    activators : List[str]
18        activators[1] = activation function of layer 1+1
19    Returns
20    -----
21    cache : Dict[Dict]
```

```

22         cache['z'][l] : array_like
23         z[l].shape = (layers[l], n)
24         cache['a'][l] : array_like
25         a[l].shape = (layers[l], n)
26     """
27     # Retrieve parameters
28     w = params['w']
29     b = params['b']
30     L = len(w) # Number of layers excluding output layer
31     n = x.shape[1]
32     # Set empty caches
33     a = {}
34     z = {}
35     # Initialize a
36     a[0] = x
37     for l in range(1, L + 1):
38         z[l], a[l] = utils.linear_activation_forward(a[l - 1], w[l], b[l], activator
39
40     cache = {'a' : a, 'z' : z}
41     return cache
42
43 def compute_cost(y, params, cache, lambda_=0.0):
44     """
45     Parameters
46     -----
47     y : array_like
48         y.shape = (layers[-1], n)
49     params : Dict[Dict[array_like]]
50         params['w'][l] : array_like
51             w[l].shape = (layers[l], layers[l-1])
52         params['b'][l] : array_like
53             b[l].shape = (layers[l], 1)
54     cache : Dict[Dict[array_like]]
55         cache['z'][l] : array_like
56             z[l].shape = (layers[l], n)
57         cache['a'][l] : array_like
58             a[l].shape = (layers[l], n)
59     lambda_ : float
60         Default: 0.0
61
62     Returns
63     -----
64     cost : float
65         The cost evaluated at y and aL
66     """
67     ## Retrieve parameters
68     n = y.shape[1]

```

```

69     a = cache['a']
70     w = params['w']
71     L = len(a)
72     aL = a[L - 1]
73
74     ## Regularization term
75     R = 0
76     for l in range(1, L):
77         R += np.sum(w[l] * w[l])
78     R *= (lambda_ / (2 * n))
79
80     ## Unregularized cost
81     J = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
82
83     ## Total Cost
84     cost = J + R
85     cost = float(np.squeeze(cost))
86     return cost
87
88 def backward_propagation(x, y, params, cache, activators, lambda_=0.0):
89     """
90     Parameters
91     -----
92     x : array_like
93         x.shape = (layers[0], n)
94     y : array_like
95         y.shape = (layers[-1], n)
96     params : Dict[Dict[array_like]]
97         params['w'][l] : array_like
98             w[l].shape = (layers[l], layers[l-1])
99         params['b'][l] : array_like
100             b[l].shape = (layers[l], 1)
101     cache : Dict[Dict[array_like]]
102         cache['a'][l] : array_like
103             a[l].shape = (layers[l], n)
104         cache['z'][l] : array_like
105             z[l].shape = (layers[l], n)
106     activators : List[str]
107         activators[l] = activation function of layer l+1
108     lambda_ : float
109         Default: 0.0
110
111     Returns
112     -----
113     grads : Dict[Dict]
114         grads['dw'][l] : array_like
115             dw[l].shape = w[l].shape

```

```

116         grads['db'][l] : array_like
117         db[l].shape = b[l].shape
118     """
119     ## Retrieve parameters
120     a = cache['a']
121     z = cache['z']
122     w = params['w']
123     n = x.shape[1]
124     L = len(z)
125
126     ## Compute deltas
127     delta = {}
128     delta[L] = a[L] - y
129     for l in reversed(range(1, L)):
130         delta[l] = utils.linear_activation_backward(delta[l + 1], z[l], w[l + 1], a
131
132     ## Compute gradients
133     dw = {}
134     db = {}
135     for l in range(1, L + 1):
136         db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
137         assert(db[l].shape == (w[l].shape[0], 1))
138         dw[l] = (1 / n) * (delta[l] @ a[l - 1].T + lambda_ * w[l])
139         assert(dw[l].shape == w[l].shape)
140     grads = {'dw' : dw, 'db' : db}
141     return grads
142
143
144 def model(x, y,
145          hidden_layer_sizes,
146          activators,
147          lambda_=0.0,
148          num_iters=1e4,
149          print_cost=False):
150     """
151     Parameters
152     -----
153     x : array_like
154         x.shape = (layers[0], n)
155     y : array_like
156         y.shape = (layers[-1], n)
157     hidden_layer_sizes : List[int]
158         The number nodes layer l = hidden_layer_sizes[l-1]
159     activators : List[str]
160         activators[l] = activation function of layer l+1
161     lambda_ : float
162         The regularization parameter

```

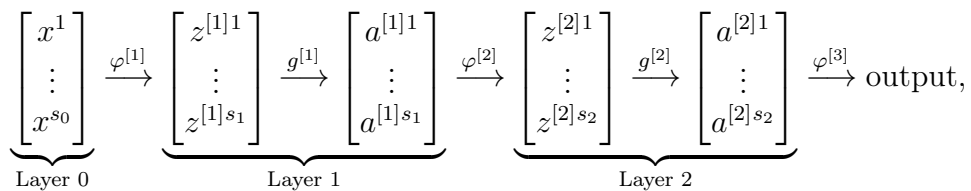
```

163         Default: 0.0
164     num_iters : int
165         Number of iterations with which our model performs gradient descent
166         Default: 10000
167     print_cost : Boolean
168         If True, print the cost every 1000 iterations
169         Default: False
170
171     Returns
172     -----
173     params : Dict[Dict]
174         params['w'][1] : array_like
175             w[1].shape = (layers[1], layers[1-1])
176         params['b'][1] : array_like
177             b[1].shape = (layers[1], 1)
178     cost : float
179         The final cost value for the optimized parameters returned
180     """
181     ## Set dimensions and Initialize parameters
182     n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
183     params = utils.initialize_parameters_random(layers)
184
185     # main gradient descent loop
186     for i in range(num_iters):
187         cache = forward_propagation(x, params, activators)
188         cost = compute_cost(y, params, cache, lambda_)
189         grads = backward_propagation(x, y, params, cache, activators, lambda_)
190         params = utils.update_parameters(params, grads)
191
192         if print_cost and i % 1000 == 0:
193             print(f'Cost_after_iteration_{i}:_{cost}')
194
195     return params, cost

```

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 numpy as np
2
3 import utils
4
5 def dropout_matrices(layers, num_examples, keep_prob):
6     """
7     Parameters
8     -----
9     layers : List[int]
10         layers[l] = number of nodes in layer l
11     num_examples : int
12         The number of training examples
13     keep_prob : List[float]
14         keep_prob[l] = The probability of keeping a node in layer l
15
16     Returns
17     -----
18     D : Dict[array_like]
19         D[l].shape = (layers[l], num_ex)
20         D[l] = a Boolean array
21     """
22     np.random.seed(1)
23     L = len(layers)
24     D = {}
25     for l in range(L - 1):
26         D[l] = np.random.rand(layers[l], num_examples)
27         D[l] = (D[l] < keep_prob[l]).astype(int)
28         assert(D[l].shape == (layers[l], num_examples))
29     return D
30
31
32
33 def forward_propagation(x, params, activators, D, keep_prob):
34     """
35     Parameters
36     -----
37     x : array_like
38         x.shape = (layers[0] n)
39     params : Dict[Dict]
40         params['w'][l] : array_like
41             w_l.shape = (layers[l], layers[l-1])
42         params['b'][l] : array_like
43             b_l.shape = (layers[l], 1)
44     activators : List[str]
45         activators[l] = activation function of layer l+1
46     D : Dict[array_like]
47         D[l].shape = (layer_dims[l], num_ex)

```

```

48         D[l] = a Boolean array
49     keep_prob : List[float]
50         keep_prob[l] = The probability of keeping a node in layer l
51
52     Returns
53     -----
54     cache : Dict[Dict]
55         cache['z'][l] : array_like
56             z[l].shape = (layers[l], n)
57         cache['a'][l] : array_like
58             a[l].shape = (layers[l], n)
59     """
60     # Retrieve parameters
61     w = params['w']
62     b = params['b']
63     L = len(w) # Number of layers including input layer
64     n = x.shape[1]
65
66     # Set empty caches
67     a = {}
68     z = {}
69     # Dropout on layer 0
70     a[0] = x
71     a[0] = a[0] * D[0]
72     a[0] /= keep_prob[0]
73     # Loop through hidden layers
74     for l in range(1, L):
75         z[l], al = utils.linear_activation_forward(a[l - 1], w[l], b[l], activators[1])
76         al = al * D[l]
77         al /= keep_prob[l]
78         z[l] = z[l]
79         a[l] = al
80
81     # Output layer
82     z[L], a[L] = utils.linear_activation_forward(a[L - 1], w[L], b[L], activators[-1])
83
84     cache = {'z' : z, 'a' : a}
85     return cache
86
87 def backward_propagation(x, y, params, cache, activators, D, keep_prob):
88     """
89     Parameters
90     -----
91     x : array_like
92         x.shape = (layers[0], n)
93     y : array_like
94         y.shape = (layers[-1], n)

```

```

95     params : Dict
96         params['w'][l] : array_like
97         w[l].shape = (layers[l], layers[l-1])
98         params['b'][l] : array_like
99         b[l].shape = (layers[l], 1)
100     cache : Dict
101         cache['a'][l] : array_like
102         a[l].shape = (layers[l], n)
103         cache['z'][l] : array_like
104         z[l].shape = (layers[l], n)
105     activators : List[str]
106         activators[l] = activation function of layer l+1
107     D : Dict[array_like]
108         D[l].shape = (layer[l], num_ex)
109         D[l] = a Boolean array
110     keep_prob : List[float]
111         keep_prob[l] = The probabilty of keeping a node in layer l
112
113     Returns
114     -----
115     grads : Dict[Dict]
116         grads['dw'][l] : array_like
117         dw[l].shape = w[l].shape
118         grads['db'][l] : array_like
119         db[l].shape = b[l].shape
120
121     """
122     ## Retrieve parameters
123     a = cache['a']
124     z = cache['z']
125     w = params['w']
126     n = x.shape[1]
127     L = len(z)
128
129     ## Compute deltas
130     delta = {}
131     delta[L] = a[L] - y
132     for l in reversed(range(1, L)):
133         deltal = utils.linear_activation_backward(delta[l + 1], z[l], w[l + 1], act:
134         deltal = deltal * D[l]
135         deltal /= keep_prob[l]
136         delta[l] = deltal
137
138     ## Compute gradients
139     dw = {}
140     db = {}
141
142     for l in range(1, L + 1):

```

```

142         db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
143         assert(db[l].shape == (w[l].shape[0], 1))
144         dw[l] = (1 / n) * delta[l] @ a[l - 1].T
145         assert(dw[l].shape == w[l].shape)
146     grads = {'dw' : dw, 'db' : db}
147     return grads
148
149 def model(x, y,
150         hidden_sizes,
151         activators,
152         keep_prob = 1.0,
153         num_iters=2500,
154         learning_rate=0.1,
155         print_cost=False):
156     """
157     Parameters
158     -----
159     Parameters
160     -----
161     x : array_like
162         x.shape = (layers[0], n)
163     y : array_like
164         y.shape = (layers[-1], n)
165     hidden_sizes : List[int]
166         The number nodes layer l = hidden_sizes[l-1]
167     activators : List[function]
168         activators[l] = activation function of layer l+1
169     keep_prob : List[float] | float
170         keep_prob[l] = The probability of keeping a node in layer l
171         keep_prob = The same probability for all input and hidden layers
172     num_iters : int
173         Number of iterations with which our model performs gradient descent
174     learning_rate : float
175         The learning rate for gradient descent
176     print_cost : Boolean
177         If True, print the cost every 1000 iterations
178
179     Returns
180     -----
181     params : Dict[Dict]
182         params['w'][l] : array_like
183             w[l].shape = (layers[l], layers[l-1])
184         params['b'][l] : array_like
185             b[l].shape = (layers[l], 1)
186     cost : float
187         The final cost value for the optimized parameters returned
188     """

```

```

189     ## Retrieve parameters
190     n, layers = utils.dim_retrieval(x, y, hidden_sizes)
191     params = utils.initialize_parameters_random(layers)
192
193     ## Expand keep_prob to a list if it's a single float
194     if isinstance(keep_prob, float):
195         keep_prob = [keep_prob] * (len(layers) - 1)
196     ## Main gradient descent loop
197     for i in range(num_iters):
198         D = dropout_matrices(layers, n, keep_prob)
199         cache = forward_propagation(x, params, activators, D, keep_prob)
200         cost = utils.compute_cost(y, cache)
201         grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
202         params = utils.update_parameters(params, grads, learning_rate)
203
204         if print_cost and i % 1000 == 0:
205             print(f'Cost_after_iteration_{i}:_{cost}')
206
207     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. See “He Initialization” and “Xavier Initialization”

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 is 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     y = 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 y, 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     y_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         y_plus, _ = f(x_plus)
35         y_minus, _ = f(x_minus)
36         y_diffs.append(y_plus - y_minus)
37     y_diffs = np.array(y_diffs).reshape(n, 1)
38     y_diffs = y_diffs / (2 * epsilon)
39
40     error = (np.linalg.norm(y_diffs - grad)
41             / (np.linalg.norm(y_diffs) + np.linalg.norm(grad)))
42     return error
```

7 Gradient Descent

So far in our implementation of gradient descent, we use the entire training set for every iteration of gradient descent. This method is called *batch gradient descent*. Gradient descent has many downfalls. Indeed, since we're typically working in a *very* high dimensional space, the majority of the critical points for our cost function are actually saddle points (these can be thought of as plateaus of the loss-manifold). These pitfalls (amongst others) are what we wish to overcome. To this end, we first consider a modification of batch gradient descent by partitioning the training set into smaller "mini-batches" and using each mini-batch recursively throughout the iterative process.

That is, suppose we have training set \mathbb{X} with $|\mathbb{X}| = n$, where n is very large (e.g., $n = 5000000$). We fix a batch size b (e.g., $b = 5000$), and partition \mathbb{X} into (e.g., 1000 distinct) mini-batches

$$\left\{ \mathbb{X}^t : 1 \leq t \leq \left\lceil \frac{n}{b} \right\rceil \right\}, \quad \mathbb{X} = \bigcup_{t=1}^{\left\lceil \frac{n}{b} \right\rceil} \mathbb{X}^t,$$

where $\left\lceil \frac{n}{b} \right\rceil$ denote the ceiling function. If we shuffle \mathbb{X} and partition during each epoch (i.e., each iteration) so our loss-manifold changes during each batch iteration within each epoch, we can then perform gradient descent in the following manner:

1. For $0 \leq i < \text{num_iters}$:
 - a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^t\}$.
 - b. For $1 \leq t \leq B$:
 - i. Perform forward propagation on \mathbb{X}^t :

$$\begin{aligned} a^{[0]} &= \mathbb{X}^t \\ z^{[\ell]} &= W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} \\ a^{[\ell]} &= g^{[\ell]}(z^{[\ell]}) \end{aligned}$$

- ii. Evaluate the cost \mathbb{J}^t on \mathbb{X}^t :

$$\mathbb{J}^t(W, b) = \frac{1}{|\mathbb{X}^t|} \sum_{(x,y) \in \mathbb{X}^t} \mathbb{L}(\hat{y}, y) + \frac{\lambda}{2|\mathbb{X}^t|} \sum_{\ell=1}^L \|W^{[\ell]}\|_F^2.$$

iii. Perform backward propagation on \mathbb{X}^t :

$$\begin{aligned}\frac{\partial \mathbb{J}^t}{\partial W^{[\ell]}} &= \frac{1}{|\mathbb{X}^t|} \delta^{[\ell]} a^{[\ell-1]T} + \frac{\lambda}{|\mathbb{X}^t|} W^{[\ell]} \\ \frac{\partial \mathbb{J}^t}{\partial b^{[\ell]}} &= \frac{1}{|\mathbb{X}^t|} \sum_{\rho \sim \mathbb{X}^t} \delta^{[\ell]}_{\rho}\end{aligned}$$

iv. Perform gradient descent:

$$\begin{aligned}W^{[\ell]} &:= W^{[\ell]} - \alpha \frac{\partial \mathbb{J}^t}{\partial W^{[\ell]}} \\ b^{[\ell]} &:= b^{[\ell]} - \alpha \frac{\partial \mathbb{J}^t}{\partial b^{[\ell]}}\end{aligned}$$

We make several remarks about mini-batch gradient descent:

- Batch gradient descent doesn't always decrease (e.g., our learning rate is too large). Mini-batch may oscillate rapidly, but the general direction should move towards a minimum.
- If $b = n$, then we fully recover batch gradient descent. This is typically too computationally expensive since we use the full training set for each iteration.
- If $b = 1$, then we recover stochastic gradient descent, i.e., we train our model on a different example during each iteration. We lose all the speed related to vectorization, since we're dealing with single examples during each iteration.
- Choose $1 < b < n$ is typically always the best solution, since it deals with both of the aforementioned problems.
- Due to the nature of a computer's internal structure, it's typically better to choose a batch size b for the form

$$b = 2^p,$$

for some $p \in \{6, 7, 8, 9, 10\}$ (usually $p < 10$).

- Choose a batch size b that ensures your computer's CPU/GPU can hold a dataset of that size.

7.1 Weighted Averages

Suppose $x_t \in \mathbb{R}^m$ is some collection of data indexed by t which we may consider a time-variable, that is, after each successive unit of time (say for example, each day), our collection adds a new data point. That is, the collection

$$\{x_t \in \mathbb{R}^m : 1 \leq t \leq T\}$$

has variable T .

Then if X is the random vector associated to x , our usual mean μ is given by

$$\mu(T) := \mathbb{E}[X] = \frac{1}{T} \sum_{t=1}^T x_t.$$

Since our collection of data is growing and evolving over time, it's reasonable in many applications to have the most recent data points affect a model more than older data points. That is, we wish to impose a “weight” on more recent data points.

One way (and likely the most trivial) to achieve such a weighing is to have only the most recent k examples affect our model. That is, for fixed $k \in \mathbb{N}$, and $t \geq k$, define the vector $\hat{x}_{t+1} \in \mathbb{R}^m$ by

$$\hat{x}_{t+1} = \frac{1}{k} \sum_{j=t-k+1}^t x_j.$$

Then \hat{x}_{t+1} represents the mean of the most recent k -examples. This may be interpreted as the “predicted-value” for x_{t+1} . This predictive model is known as a *simple moving average*, or *SMA*.

The simple moving average satisfies our weight requirement of focusing more on the most recent data, however, older data, though being less relevant, should still affect our model, but in a reduced form. The simple model does not satisfy this more refined requirement. Let's modify the simple model as follows: Fix $\beta_1 \in [0, 1)$ and we initialize a $V_0 = 0 \in \mathbb{R}^m$, and define recursively the vector $V_t \in \mathbb{R}^m$ given by

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1) x_t.$$

We claim that V_t can be interpreted as the next predicted value \hat{x}_{t+1} . Indeed,

expanding our recursive definition

$$\begin{aligned}
V_t &= \beta_1 V_{t-1} + (1 - \beta_1)x_t \\
&= \beta_1(\beta_1 V_{t-2} + (1 - \beta_1)x_{t-1}) + (1 - \beta_1)x_t \\
&= \beta_1^2 V_{t-2} + (1 - \beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^2(\beta_1 V_{t-3} + (1 - \beta_1)x_{t-2}) + (1 - \beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^3 V_{t-3} + (1 - \beta_1)(\beta_1^2 x_{t-2} + \beta_1 x_{t-1} + x_t) \\
&\vdots \\
&= \beta_1^t V_0 + (1 - \beta_1) \sum_{j=0}^{t-1} \beta_1^j x_{t-j} \\
&= (1 - \beta_1) \sum_{j=0}^{t-1} \beta_1^j x_{t-j}.
\end{aligned}$$

Moreover, if we define a probability distribution \mathbb{P} as given by

$$\mathbb{P}(X = x_j) = (1 - \beta_1)\beta_1^j,$$

then we immediately see that V_t is the weighted-average over the last t -days, and hence may be interpreted as the predicted-value \hat{x}_{t+1} as desired. Finally, since

$$1 - \beta_1 = \frac{1}{\frac{1}{1-\beta_1}},$$

we may interpret $\frac{1}{1-\beta_1}$ as the size of the relevant sampling, i.e., V_t is the average of x over the previous $\frac{1}{1-\beta_1}$ days (assuming our time-units are measured in days). This predictive model is known as an *exponentially moving average*, or *EMA*.

Remark 7.1. *We note that since we initialize our EMA with $V_0 = 0$, that our predictive model is very bad for small t . This usually is irrelevant for many models, but if we need to correct for bias, we may make the modification of*

$$V_t = \frac{\beta_1 V_{t-1} + (1 - \beta_1)x_t}{1 - \beta_1^t}.$$

Indeed, since $\beta_1 \in [0, 1)$, we note that

$$\begin{aligned}
\frac{1}{1 - \beta_1} &= \sum_{j=0}^{\infty} \beta_1^j \\
&= \sum_{j=t}^{\infty} \beta_1^j + \sum_{j=0}^{t-1} \beta_1^j \\
&= \beta_1^t \sum_{j=0}^{\infty} \beta_1^j + \sum_{j=0}^{t-1} \beta_1^j \\
&= \frac{\beta_1^t}{1 - \beta_1} + \sum_{j=0}^{t-1} \beta_1^j,
\end{aligned}$$

and so

$$\sum_{j=0}^{t-1} \beta_1^j = \frac{1 - \beta_1^t}{1 - \beta_1}.$$

We then see that

$$\begin{aligned}
V_t &= \frac{\beta_1 V_{t-1} + (1 - \beta_1)x_t}{1 - \beta_1^t} \\
&= \frac{(1 - \beta_1) \sum_{j=0}^{t-1} \beta_1^j x_{t-j}}{1 - \beta_1^t} \\
&= \frac{\sum_{j=0}^{t-1} \beta_1^j x_{t-j}}{\sum_{j=0}^{t-1} \beta_1^j},
\end{aligned}$$

which is the explicit definition of a weighted-average.

7.2 Gradient Descent with Momentum

Gradient descent has an issue with potentially plateauing during areas with a flat gradient, or bouncing around drastically before arriving at a minimum. One reason for this is that each iterative step only depends on the previous value of the gradient (or rather, the most recently updated parameter). The algorithm doesn't see larger trends, and so this leads to give our algorithm more history of the movements. We do this by using EMA.

We first recall our gradient descent algorithm:

1. We initialize $W^{\{0\}}$ and $b^{\{0\}}$.

2. For $0 \leq i < \text{num_iters}$:
 - a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$.
 - b. For $1 \leq k \leq B$:
 - i. Apply forward propagation on \mathbb{X}^k .
 - ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
 - iii. Apply backward propagation on \mathbb{X}^k to obtain

$$\frac{\partial \mathbb{J}^{\{t\}}}{\partial W}, \quad \frac{\partial \mathbb{J}^{\{t\}}}{\partial b}.$$

- iv. We update parameters

$$W^{\{t\}} = W^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}^{\{t\}}}{\partial W}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}^{\{t\}}}{\partial b}$$

Using this formulation of gradient descent, we insert EMA applied to the sequences of gradients depending on the iteration $t := i + k$. That is, we have the following algorithm:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize $V_W^{\{0\}} = V_b^{\{0\}} = 0$. Fix a momentum hyper-parameter $\beta_1 \in [0, 1)$.
2. For $0 \leq i < \text{num_iters}$:
 - a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$.
 - b. For $1 \leq k \leq B$:
 - i. Apply forward propagation on \mathbb{X}^k .
 - ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
 - iii. Apply backward propagation on \mathbb{X}^k to obtain

$$\frac{\partial \mathbb{J}^{\{t\}}}{\partial W}, \quad \frac{\partial \mathbb{J}^{\{t\}}}{\partial b}.$$

- iv. Define

$$V_W^{\{t\}} = \beta_1 V_W^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}^{\{t\}}}{\partial W}$$

$$V_b^{\{t\}} = \beta_1 V_b^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}^{\{t\}}}{\partial b}$$

v. We update parameters

$$\begin{aligned} W^{\{t\}} &= W^{\{t-1\}} - \alpha V_W^{\{t\}} \\ b^{\{t\}} &= b^{\{t-1\}} - \alpha V_b^{\{t\}} \end{aligned}$$

7.3 Root Mean Squared Propagation (RMSProp)

One of the main drawbacks to gradient descent with momentum is the uniformity of the modification regardless of the direction. That is, suppose our desired minimum is in the \vec{b} direction, but the gradient $\partial_b \mathbb{J}$ is small while the gradient $\partial_W \mathbb{J}$ is large. As a result, our steps will oscillate wildly in the \vec{w} direction, while moving very slowly in the \vec{b} direction to our desired minimum. This as a whole can be very computationally slow, and is undesired.

The main idea for fixing these oscillatory issues is have a variable learning rate α which also depends on the direction. That is, if $\partial_W \mathbb{J}$ is large, and not in our desired direction of motion, we would like our update for W to be small, and vice-versa if $\partial_b \mathbb{J}$ is small. Moreover, we wish to exaggerate the magnitudes of these vectors so we ensure our algorithm works efficiently. That is, we relate some vector S via

$$S \sim \frac{\partial \mathbb{J}^2}{\partial W},$$

where we're taking that Hadamard-square (i.e., component-wise product with itself). Then we perform step via

$$W = W - \alpha \frac{1}{\sqrt{S}} \odot \frac{\partial \mathbb{J}}{\partial W},$$

where where taking the Hadamard-root. Note that this root is necessary for our update to make sense (consider the units involved in such an equation), but it does introduce the potential to divide by zero (which we'll fix by a small ϵ). Moreover, we would like use the history of gradients as in EMA to further our refinement of the descent algorithm. To this end, we have the following *RMSProp algorithm*:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize $S_W^{\{0\}} = S_b^{\{0\}} = 0$. Fix a momentum $\beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small ($\epsilon = 10^{-8}$ is a good starting point).
2. For $0 \leq i < \text{num_iter}$:

- a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$
- b. For $1 \leq k \leq B$:
 - i. Apply forward propagation on \mathbb{X}^k .
 - ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
 - iii. Apply backward propagation on \mathbb{X}^k to obtain

$$\frac{\partial \mathbb{J}^{\{t\}}}{\partial W} \quad , \quad \frac{\partial \mathbb{J}^{\{t\}}}{\partial b} \quad .$$

- iv. Define

$$S_W^{\{t\}} = \beta_2 S_W^{\{t-1\}} + (1 - \beta_2) \left(\frac{\partial \mathbb{J}^{\{t\}}}{\partial W} \right)^2$$

$$S_b^{\{t\}} = \beta_2 S_b^{\{t-1\}} + (1 - \beta_2) \left(\frac{\partial \mathbb{J}^{\{t\}}}{\partial b} \right)^2$$

- v. Update parameters via

$$W^{\{t\}} = W^{\{t-1\}} - \alpha \frac{\frac{\partial \mathbb{J}^{\{t\}}}{\partial W}}{\sqrt{S_W^{\{t\}} + \epsilon}}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\frac{\partial \mathbb{J}^{\{t\}}}{\partial b}}{\sqrt{S_b^{\{t\}} + \epsilon}}$$

7.4 Adaptive Moment Estimation: The Adam Algorithm

We first note that with the momentum algorithm utilizing the EMA as it does, that it is an algorithm of the first moment (i.e., the mean of the gradients). Similarly, with RMSProp utilizing the square of the gradient as it does, we say it is an algorithm of the second moment (i.e., the uncentered variance of the gradients). Our goal is to utilize both gradient descent with momentum and RMSProp simultaneously to optimize our parameters. This combination of algorithms is called the *Adam algorithm* and is implemented as follows:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize $V_W^{\{0\}} = V_b^{\{0\}} = 0$ and $S_W^{\{0\}} = S_b^{\{0\}} = 0$. Fix our constants of momenta $\beta_1, \beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small.

2. For $0 \leq i < \text{num_iters}$:

- a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$
- b. For $1 \leq k \leq B$:
 - i. Apply forward propagation on \mathbb{X}^k .
 - ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
 - iii. Apply backward propagation on \mathbb{X}^k to obtain

$$\frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \quad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$

iv. Define

$$\begin{aligned} V_W^{\{t\}} &= \beta_1 V_W^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial W}^{\{t\}}, \\ V_b^{\{t\}} &= \beta_1 V_b^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}, \end{aligned}$$

and define

$$\begin{aligned} S_W^{\{t\}} &= \beta_2 S_W^{\{t-1\}} + (1 - \beta_2) \left(\frac{\partial \mathbb{J}}{\partial W}^{\{t\}} \right)^2, \\ S_b^{\{t\}} &= \beta_2 S_b^{\{t-1\}} + (1 - \beta_2) \left(\frac{\partial \mathbb{J}}{\partial b}^{\{t\}} \right)^2. \end{aligned}$$

v. Utilize bias correction via:

$$\begin{aligned} \hat{V}_W^{\{t\}} &= \frac{V_W^{\{t\}}}{1 - \beta_1^t} \\ \hat{V}_b^{\{t\}} &= \frac{V_b^{\{t\}}}{1 - \beta_1^t} \\ \hat{S}_W^{\{t\}} &= \frac{S_W^{\{t\}}}{1 - \beta_2^t} \\ \hat{S}_b^{\{t\}} &= \frac{S_b^{\{t\}}}{1 - \beta_2^t} \end{aligned}$$

vi. Update the parameters:

$$W^{\{t\}} = W^{\{t-1\}} - \alpha \frac{\hat{V}_W^{\{t\}}}{\sqrt{\hat{S}_W^{\{t\}} + \epsilon}}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\hat{V}_b^{\{t\}}}{\sqrt{\hat{S}_b^{\{t\}} + \epsilon}}$$

We note that though we may still need to tune the hyper-parameter α , the hyper-parameters β_1, β_2 and ϵ typically work quite well with default values of

$$\beta_1 = 0.9, \quad \beta_2 = 0.999, \quad \epsilon = 10^{-8}.$$

7.5 Learning Rate Decay

Finally, one further method we may utilize in our optimization problem, is the idea of slowly reducing our learning rate α . That is, if i is our epoch iteration, and $\eta > 0$ is a fixed decay rate, we can define new learning rates in many ways. That is, for $\alpha = \alpha(i)$ we can define

•

$$\alpha(i) = \frac{1}{1 + \eta i} \alpha_0,$$

•

$$\alpha(i) = \alpha_0 \eta^i,$$

•

$$\alpha(i) = \frac{\eta}{\sqrt{i}} \alpha_0.$$

One could also implement a “manual decay”, but this should only be used under ideal circumstances.

7.5.1 Python Implementation

```
1 import copy
2
3 import numpy as np
4 from sklearn.utils import shuffle
5
6 import utils
```

```

7
8 def get_batches(x, y, b):
9     """
10    Parameters
11    -----
12    x : array_like
13        x.shape = (m, n)
14    y : array_like
15        y.shape = (k, n)
16    b : int
17
18    Returns
19    -----
20    batches : List[Dict]
21        batches[i]['x'] : array_like
22            x.shape = (m, b) # except last batch
23            y.shape = (k, b) # except last batch
24
25    """
26    m, n = x.shape
27    ## Shuffle the data
28    x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
29    x = x.T
30    y = y.T
31
32    B = int(np.ceil(n / b))
33    batches = []
34    for i in range(B):
35        x_temp = x[:,(b * i):(b * (i + 1))]
36        y_temp = y[:,(b * i):(b * (i + 1))]
37        batches.append({'x' : x_temp, 'y' : y_temp})
38    # Slicing automatically ends at the end of
39    # the list if the stop is outside the index
40    return batches
41
42 def initialize_momenta(layers):
43     """
44    Parameters
45    -----
46    layers : List[int]
47        layers[l] = # nodes in layer l
48    Returns
49    -----
50    v : Dict[Dict[array_like]]
51    s : Dict[Dict[array_like]]
52    """
53    vw = {}

```

```

54     vb = {}
55     sw = {}
56     sb = {}
57     for l in range(1, len(layers)):
58         vw[l] = np.zeros((layers[l], layers[l - 1]))
59         sw[l] = np.zeros((layers[l], layers[l - 1]))
60         vb[l] = np.zeros((layers[l], 1))
61         sb[l] = np.zeros((layers[l], 1))
62
63     v = {'w' : vw, 'b' : vb}
64     s = {'w' : sw, 'b' : sb}
65
66     return v, s
67
68 def learning_rate_decay(epoch, learning_rate=0.01, decay_rate=0.0):
69     """
70     Parameters
71     -----
72     epoch : int
73     learning_rate : float
74         Default: 0.01
75     decay_rate : float
76         Default: 0.0 - Returns a constant learning_rate
77
78     Returns
79     -----
80     learning_rate : float
81     """
82     learning_rate = (1 / (1 + epoch * decay_rate)) * learning_rate
83     return learning_rate
84
85 def corrected_momentum(v, grads, update_iter, beta1=0.0):
86     """
87     Parameters
88     -----
89     v : Dict[Dict[array_like]]
90         v['w'][l].shape = w[l].shape
91         v['b'][l].shape = b[l].shape
92     grads : Dict[Dict]
93         grads['w'][l] : array_like
94             dw[l].shape = w[l].shape
95         grads['b'][l] : array_like
96             db[l].shape = b[l].shape
97     update_iter : int
98     beta1 : float
99         Default: 0.0 - Returns grads
100         Usual: 0.9

```

```

101
102 Returns
103 -----
104 v : Dict[Dict[array_like]]
105     v['w'][1].shape = dw[1].shape
106     v['b'][1].shape = db[1].shape
107     """
108     ## Retrieve velocities and gradients
109     vw = v['w']
110     vb = v['b']
111     dw = grads['w']
112     db = grads['b']
113     L = len(dw)
114
115     for l in range(1, L + 1):
116         vw[l] = beta1 * vw[l] + (1 - beta1) * dw[l]
117         vw[l] /= (1 - beta1 ** update_iter)
118         assert(vw[l].shape == dw[l].shape)
119         vb[l] = beta1 * vb[l] + (1 - beta1) * db[l]
120         vb[l] /= (1 - beta1 ** update_iter)
121         assert(vb[l].shape == db[l].shape)
122
123     v = {'w' : vw, 'b' : vb}
124     return v
125
126 def corrected_rmsprop(s, grads, update_iter, beta2=0.999):
127     """
128     Parameters
129     -----
130     s : Dict[Dict[array_like]]
131         s['w'][1].shape = w[1].shape
132         s['b'][1].shape = b[1].shape
133     grads : Dict[Dict]
134         grads['w'][1] : array_like
135             dw[1].shape = w[1].shape
136         grads['b'][1] : array_like
137             db[1].shape = b[1].shape
138     update_iter : int
139     beta2 : float
140         Default: 0.999
141
142     Returns
143     -----
144     s : Dict[Dict[array_like]]
145         s['w'][1].shape = w[1].shape
146         s['b'][1].shape = b[1].shape
147     """

```

```

148     ## Retrieve accelerations and gradients
149     sw = s['w']
150     sb = s['b']
151     dw = grads['w']
152     db = grads['b']
153     L = len(dw)
154
155     for l in range(1, L + 1):
156         sw[l] = beta2 * sw[l] + (1 - beta2) * (dw[l] * dw[l])
157         sw[l] /= (1 - beta2 ** update_iter)
158         assert(sw[l].shape == dw[l].shape)
159         sb[l] = beta2 * sb[l] + (1 - beta2) * (db[l] * db[l])
160         sb[l] /= (1 - beta2 ** update_iter)
161         assert(sb[l].shape == db[l].shape)
162
163     s = {'w' : sw, 'b' : sb}
164     return s
165
166
167 def update_parameters_adam(params, grads, epoch, batch_iter, v, s, momenta=[1e-8, 0.
168     """
169     Parameters
170     -----
171     params : Dict[Dict]
172         params['w'][l] : array_like
173             w[l].shape = (layers[l], layers[l-1])
174         params['b'][l] : array_like
175             b[l].shape = (layers[l], 1)
176     grads : Dict[Dict]
177         grads['dw'][l] : array_like
178             dw[l].shape = w[l].shape
179         grads['db'][l] : array_like
180             db[l].shape = b[l].shape
181     epoch : int
182     batch_iter : int
183     learning_rate : float
184         Default: 0.01
185     momenta : List[float]
186         momenta[0] = epsilon
187             Default: 10^{-8}
188         momenta[1] = beta_1
189             Default: 0.9
190         momenta[2] = beta_2
191             Default: 0.999
192
193     Returns
194     -----

```

```

195     params : Dict[Dict]
196         params['w'][l] : array_like
197         w[l].shape = (layers[l], layers[l-1])
198         params['b'][l] : array_like
199         b[l].shape = (layers[l], 1)
200     """
201     update_iter = epoch + batch_iter
202     ## Retrieve parameters
203     w = copy.deepcopy(params['w'])
204     b = copy.deepcopy(params['b'])
205     L = len(w)
206
207     ## Update velocities and accelerations
208     v = corrected_momentum(v, grads, update_iter, momenta[1])
209     vw = v['w']
210     vb = v['b']
211     s = corrected_rmsprop(s, grads, update_iter, momenta[2])
212     sw = s['w']
213     sb = s['b']
214
215     ## Update learning rate
216     learning_rate = learning_rate_decay(epoch, alpha0, decay_rate)
217
218     ## Perform update
219     for l in range(1, L + 1):
220         w[l] = w[l] - learning_rate * vw[l] / (np.sqrt(sw[l]) + momenta[0])
221         b[l] = b[l] - learning_rate * vb[l] / (np.sqrt(sb[l]) + momenta[0])
222
223     params = {'w' : w, 'b' : b}
224     return params
225
226 def model(x, y,
227         hidden_layer_sizes,
228         activators,
229         batch_size,
230         lambda_=0.0,
231         num_iters=10000,
232         print_cost=False):
233     """
234     Parameters
235     -----
236     x : array_like
237         x.shape = (layers[0], n)
238     y : array_like
239         y.shape = (layers[-1], n)
240     hidden_layer_sizes : List[int]
241         The number nodes layer l = hidden_layer_sizes[l-1]

```

```

242     activators : List[str]
243         activators[l] = activation function of layer l+1
244     batch_size : int
245     lambda_ : float
246         The regularization parameter
247         Default: 0.0
248     num_iters : int
249         Number of iterations with which our model performs gradient descent
250         Default: 10000
251     print_cost : Boolean
252         If True, print the cost every 1000 iterations
253         Default: False
254
255     Returns
256     -----
257     params : Dict[Dict]
258         params['w'][l] : array_like
259             w[l].shape = (layers[l], layers[l-1])
260         params['b'][l] : array_like
261             b[l].shape = (layers[l], 1)
262     cost : float
263         The final cost value for the optimized parameters returned
264     """
265     n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
266     params = utils.initialize_parameters_random(layers)
267     v, s = initialize_momenta(layers)
268
269
270     ## main descent loop
271     for i in range(num_iters):
272         batches = get_batches(x, y, batch_size)
273         ## batch loop
274         batch_iter = 1
275         cost = 0
276         for batch in batches:
277             x = batch['x']
278             y = batch['y']
279             cache = utils.forward_propagation(x, params, activators)
280             cost += utils.compute_cost(y, params, cache)
281             grads = utils.backward_propagation(x, y, params, cache, activators)
282             params = update_parameters_adam(params,
283                                             grads,
284                                             i,
285                                             batch_iter,
286                                             v,
287                                             s,
288                                             momenta=[1e-8, 0.9, 0.999],

```

```
289                                     learning_rate=0.01,
290                                     decay_rate = 0.0)
291         batch_iter += 1
292
293         if print_cost and i % 1000 == 0:
294             print(f'Cost_after_iteration_{i}:_{cost}')
295
296     return params, cost
```


8 Tuning Hyper-Parameters

Suppose that we have the dataset \mathbb{D} with the usual partition of

$$\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}.$$

Furthermore, suppose we impose a neural network architecture which has a collection of hyper-parameters (reabeled as):

$$\eta_1, \eta_2, \dots, \eta_K.$$

The naive method of hyper-parameter tuning would instinctively be something of the form: Let $[d_i, d_i + k_i \Delta_i]$ denote an interval for which we require

$$\eta_i \in [d_i, d_i + k_i \Delta_i],$$

with an even-partition of

$$d_i < d_i + \Delta_i < d_i + 2\Delta_i < \dots < d_i + k_i \Delta_i,$$

of length Δ_i . This collection forms a “grid” in \mathbb{R}^K for which each point of the grid gives us a full collection of hyper-parameters which we can then use to train our model. However, if certain hyper-parameters do not affect our model’s accuracy very much, we’ve added at least a full dimension of validation which is not needed. A more randomized approach would be best to determine such a hyper-parameter characterization must faster. Thus a random collection of points H_i for which we constrain $\eta_i \in H_i$.

How should we implement this set H_i ? Suppose for example, we wish to find

$$\eta_i \in [0.0001, 1],$$

but the majority of the random points will likely be in $[0.1, 1]$. Suppose we partition the interval

$$\begin{aligned} [0.0001, 1] &= 0.0001 < 0.001 < 0.01 < 0.1 < 1 \\ &= 10^{-4} < 10^{-3} < 10^{-2} < 10^{-1} < 10^0. \end{aligned}$$

This suggests we obtain a distribution of points using a logarithmic (in base 10) scale. Indeed, let

$$p \in [0, 1],$$

be a random point. Then letting $r = -4p \in [-4, 0]$, we obtain another random point, and let

$$H_i = \{10^{-4p} : p \in \text{rand}([0, 1])\},$$

for some prescribed set-cardinality. This allows to choose more appropriately scaled-options for our hyper-parameters.

Remark 8.1. *Suppose we're using exponentially moving averages and have a hyper-parameter $\beta_1 \in [0, 1)$. If we do not use a log-scale, then the sensitivity of our model with respect to β_1 when $\beta_1 \approx 1$ is very strong. Indeed, we recall that when $\beta_1 = 0.999$, this corresponds to averaging over the previous 1000 days. And if we change β_1 slightly to*

$$\beta_1 = 0.9995,$$

then we've changed the interpretation of our model to the previous 2000 days. A subtle change for β_1 , but a drastic change to our model. The log-scale fixes this issue immediately.

We finally note that our hyper-parameters can become *stale* over time. That is, suppose we've trained a neural network, and tuned the hyper-parameters to allow an acceptable accuracy for our model. As the model refines over time, with more data being inserted to train on, it's import to re-test our hyper-parameters to make sure our model hasn't opened up to a better choice of one (or some or all) of the hyper-parameters we've previously tuned.

8.0.1 Python Implementation

```

1 def hyperparameter_scale(k, p):
2     """
3     Parameters
4     -----
5     k : int
6         The number random points to generate
7     p : int
8         The smallest magnitude for our log-scale
9
10    Returns
11    -----
12    hypers : List[float]
13        The list of hyper-parameters with which to tune
14    """
15    hypers = []
16    for _ in range(k):
17        r = p * np.random.rand()
18        hypers.append(10 ** r)
19    return hypers

```

9 Batch Normalization

We recall feature-normalization: Suppose $x \in \mathbb{R}^{m \times n}$ is some training data. Let

$$\mu = \mathbb{E}[X], \quad \sigma^2 = \mathbb{E}[(X - \mu)^2],$$

denote the mean and variance of the random-vector representation X of x , respectively. Then we consider the map

$$x_j \mapsto \frac{x_j - \mu}{\sigma} =: \hat{x}_j,$$

to be the *normalization* of x_j .

This definition is so “vanilla”, that it should be clear that this can be easily applied to each hidden-layer of a neural network as well. However, we first note that there is an ambiguous choice amongst the implementation, namely, do we normalize $z^{[\ell]}$ or $a^{[\ell]}$, i.e., does normalization occur before or after we compute the activation unit. It seems more common to apply normalization to $z^{[\ell]}$, so that is what we do here without further mention of this choice.

Moreover, let $\gamma, \beta \in \mathbb{R}^m$, if we consider the map

$$\hat{x}_j \mapsto \gamma \odot \hat{x}_j + \beta := \tilde{x}_j,$$

we can see fairly trivially that we can recover x_j , indeed, let $\gamma = \sigma$ and $\beta = \mu$, and hence

$$\begin{aligned} \tilde{x}_j &= \gamma \odot \hat{x}_j + \beta \\ &= \gamma \odot \frac{x_j - \mu}{\sigma} + \beta \\ &= x_j - \mu\beta \\ &= x_j \end{aligned}$$

as desired. Moreover, we see that we can actually control what mean and variance we wish to impose on our input-vectors x . Indeed, consider

$$\begin{aligned} \mathbb{E}[\gamma \odot X + \beta] &= \frac{1}{n} \sum_{j=1}^n (\gamma \odot x_j + \beta) \\ &= \gamma \odot \mathbb{E}[X] + \beta \\ &= \gamma \odot \mu + \beta, \end{aligned}$$

and so if X was normalized, we the new mean would be given by β . Similarly,

$$\begin{aligned}\mathbb{E}[(\gamma \odot X + \beta)^2] &= \frac{1}{n} \sum_{j=1}^n (\gamma \odot x_j + \beta)^2 \\ &= \frac{1}{n} \sum_{j=1}^n (\gamma^2 \odot x_j^2 + 2\beta \odot \gamma \odot x_j + \beta^2) \\ &= \gamma^2 \odot \sigma^2 + 2\beta \odot \gamma \odot \mu + \beta^2,\end{aligned}$$

and so if X was normalized, we see the new variance would be given by $\gamma^2 + \beta^2$. Thus, we see that by composition, the act of normalization can be characterized by the new parameters γ and β , and is mathematically-superfluous to consider both, but for computational considerations and algorithmic stability it shall be beneficial to keep both. That is, suppose we're training on some batch \mathbb{X}^k given a random-vector representation $Z^{[\ell]}$ in layer- ℓ , and parameters $\gamma^{[\ell]}, \beta^{[\ell]} \in \mathbb{R}^{m_\ell}$ and some $\epsilon > 0$, arbitrarily small and prescribed for numerical stability, we define the *batch-normalization* map $BN_{\gamma^{[\ell]}, \beta^{[\ell]}} : \mathbb{R}^{m_\ell} \rightarrow \mathbb{R}^{m_\ell}$ given by the compositional-map

$$\begin{aligned}z^{[\ell]} &\mapsto \frac{1}{|\mathbb{X}^k|} \sum_{x \in \mathbb{X}^k} z^{[\ell]} =: \mu^{[\ell]}; \\ (z^{[\ell]}, \mu^{[\ell]}) &\mapsto \frac{1}{|\mathbb{X}^k|} \sum_{x \in \mathbb{X}^k} (z^{[\ell]} - \mu^{[\ell]})^2 =: \sigma^{[\ell]2}; \\ (z^{[\ell]}, \mu^{[\ell]}, \sigma^{[\ell]}, \epsilon) &\mapsto \frac{z^{[\ell]} - \mu^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}} =: \hat{z}^{[\ell]}; \\ (\hat{z}^{[\ell]}, \gamma^{[\ell]}, \beta^{[\ell]}) &\mapsto \gamma^{[\ell]} \odot \hat{z}^{[\ell]} + \beta^{[\ell]} =: \tilde{z}^{[\ell]}.\end{aligned}$$

Suppose we have an L -layer neural network, each layer with m_ℓ nodes, and we focus on the ℓ -th layer specifically to expand:

$$\underbrace{\dots \xrightarrow{\varphi^{[\ell]}} \begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_\ell} \end{bmatrix} \xrightarrow{BN_{\gamma^{[\ell]}, \beta^{[\ell]}}} \begin{bmatrix} \tilde{z}^{[\ell]1} \\ \vdots \\ \tilde{z}^{[\ell]m_\ell} \end{bmatrix} \xrightarrow{g^{[\ell]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_\ell} \end{bmatrix} \xrightarrow{\varphi^{[\ell+1]}} \dots}_{\text{Layer } \ell}$$

The procedure for forward propagation should be immediately obvious from the closer look at layer- ℓ . For backward propagation, we compute derivatives

in coordinates

$$\begin{aligned}\frac{\partial \tilde{z}^{[\ell]i}}{\partial z^{[\ell]\nu}} &= \frac{\gamma^{[\ell]i}}{\sigma^{[\ell]i}} \delta_\nu^i \\ \frac{\partial \tilde{z}^{[\ell]i}}{\partial \gamma^{[\ell]\nu}} &= \frac{z^{[\ell]i} - \mu^{[\ell]i}}{\sigma^{[\ell]i}} \delta_\nu^i \\ \frac{\partial \tilde{z}^{[\ell]i}}{\partial \beta^{[\ell]\nu}} &= \end{aligned}$$