

Neural Networks

Matt R

June 3, 2022

Contents

I	Neural Networks and Deep Learning	4
1	Logistic Regression	5
1.1	The Gradient	6
1.2	Vectorization in Python	7
2	Neural Networks: A Single Hidden Layer	12
2.1	Backward Propagation	14
2.2	Activation Functions	18
2.2.1	The Sigmoid Function	18
2.2.2	The Hyperbolic Tangent Function	19
2.2.3	The Rectified Linear Unit Function	19
2.2.4	The Softmax Function	20
2.3	Binary Classification - An Example	21
2.3.1	Random Initialization	22
2.4	Vectorization in Python	23
3	Deep Neural Networks	30
3.1	Backward Propagation	30
3.2	Vectorization in Python	31
3.3	Better Backpropagation	38
II	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization	45

4	Training, Development and Test Sets	46
4.1	Python Implementation	48
5	Regularization	49
5.1	Python Implementation	50
5.2	(Inverted) Dropout Regularization	54
5.2.1	Python Implementation	55
5.3	Data Augmentation	60
5.4	Early Stopping	60
6	Gradients and Numerical Remarks	61
6.1	Numerical Gradient Checking	61
6.2	Python Implementation	62
7	Gradient Descent	63
7.1	Weighted Averages	65
7.2	Gradient Descent with Momentum	67
7.3	Root Mean Squared Propagation (RMSProp)	69
7.4	Adaptive Moment Estimation: The Adam Algorithm	70
7.5	Learning Rate Decay	72
7.6	Python Implementation	72
8	Tuning Hyper-Parameters	80
8.1	Python Implementation	81
9	Batch Normalization	82
9.1	Backward Propagation	84
9.2	Inferencing	90
9.3	Algorithm Outline	91
9.4	Better Backpropagation	93
10	Multi-Class Softmax Regression	97
III	Convolutional Neural Networks	101
11	An Introduction to Convolutions	102
11.1	Cross-Correlation	102
11.2	Convolution with Padding	104
11.3	Strided Convolution	106

11.4 Strided Convolutions with Padding	107
11.5 Convolutions Over Volumes	108
11.6 Multiple Filters	109
12 Convolutional Networks	110
12.1 Convolutional Layers (conv)	110
12.2 Pooling Layers (pool)	111
12.2.1 Max Pooling	111
12.2.2 Average Pooling	111
12.3 A Convolutional Network	112
12.4 Backpropagation	113
Appendix A utils.py	115
Appendix B activators.py	129
Appendix C The Normalization Operator	131
C.1 The Normalization Operator v.2	135
References	140

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

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

1.2 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_iters)
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 $x \in \mathbb{R}^{m_0 \times n}$ and $y \in \{0, 1\}^{1 \times n}$. Usually with logistic regression we have the following type of structure:

$$[x^1, \dots, x^{m_0}] \xrightarrow{\varphi} [z] \xrightarrow{g} [a] \xrightarrow{=} \hat{y},$$

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:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \\ \vdots \\ z^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{g^{[2]}} \underbrace{\begin{bmatrix} a^{[2]} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

where

$$\begin{aligned} \varphi^{[1]} : \mathbb{R}^{m_0} &\rightarrow \mathbb{R}^{m_1}, & \varphi^{[1]}(x) &= W^{[1]}x + b^{[1]}, \\ \varphi^{[2]} : \mathbb{R}^{m_1} &\rightarrow \mathbb{R}, & \varphi^{[2]}(x) &= W^{[2]}x + b^{[2]}, \end{aligned}$$

and $W^{[1]} \in \mathbb{R}^{m_1 \times m_0}$, $W^{[2]} \in \mathbb{R}^{1 \times m_1}$, $b^{[1]} \in \mathbb{R}^{m_1}$, $b^{[2]} \in \mathbb{R}$, and $g^{[\ell]}$ is a *broad-casted* 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:

$$\begin{aligned} \varphi^{[1]} : \mathbb{R}^{m_0} &\rightarrow \mathbb{R}^{m_1}, & d\varphi^{[1]} : T\mathbb{R}^{m_0} &\rightarrow T\mathbb{R}^{m_1}, \\ z^{[1]} = \varphi^{[1]}(x) &= W^{[1]}x + b^{[1]}, & d\varphi_x^{[1]}(v) &= W^{[1]}v; \end{aligned}$$

$$\begin{aligned}
g^{[1]} : \mathbb{R}^{m_1} &\rightarrow \mathbb{R}^{m_1}, & dg^{[1]} : T\mathbb{R}^{m_1} &\rightarrow T\mathbb{R}^{m_1}, \\
a^{[1]} &= g^{[1]}(z^{[1]}), & \frac{\partial a^{[1]\mu}}{\partial z^{[1]\nu}} &= \delta_\nu^\mu g^{[1]'}(z^{[1]\mu});
\end{aligned}$$

$$\begin{aligned}
\varphi^{[2]} : \mathbb{R}^{m_1} &\rightarrow \mathbb{R}^{m_2}, & d\varphi^{[2]} : T\mathbb{R}^{m_1} &\rightarrow T\mathbb{R}^{m_2}, \\
z^{[2]} &= \varphi^{[2]}(a^{[1]}) = W^{[2]}a^{[1]} + b^{[2]}, & d\varphi_{a^{[2]}}^{[2]}(v) &= W^{[2]}v;
\end{aligned}$$

$$\begin{aligned}
g^{[2]} : \mathbb{R}^{m_2} &\rightarrow \mathbb{R}^{m_2}, & dg^{[2]} : T\mathbb{R}^{m_2} &\rightarrow T\mathbb{R}^{m_2}, \\
a^{[2]} &= g^{[2]}(z^{[2]}), & \frac{\partial a^{[2]\mu}}{\partial z^{[2]\nu}} &= \delta_\nu^\mu g^{[2]'}(z^{[2]\mu}).
\end{aligned}$$

That is, given an input $x \in \mathbb{R}^{m_0}$, we get a predicted value $\hat{y} \in \mathbb{R}^{m_2}$ 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 Backward Propagation

backPropDerivation

Since we wish to optimize our model with respect to our parameter $W^{[\ell]}$ and $b^{[\ell]}$, we consider a generic loss function $\mathbb{L} : \mathbb{R}^{m_2} \times \mathbb{R}^{m_2} \rightarrow \mathbb{R}$, $\mathbb{L}(\hat{y}, y)$, and by acknowledging the potential abuse of notation, we assume y is fixed, and consider the aforementioned as a function of a single-variable

$$\mathbb{L}_y : \mathbb{R}^{m_2} \rightarrow \mathbb{R}, \quad \mathbb{L}_y(\hat{y}) = \mathbb{L}(\hat{y}, y).$$

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

$$F : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_0} \rightarrow \mathbb{R}$$

given by

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1}_{\mathbb{R}^{m_2 \times m_1}} \times \mathbb{1}_{\mathbb{R}^{m_2}} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

We first introduce an error term $\delta^{[2]} \in \mathbb{R}^{m_2}$ 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

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

$$\begin{aligned}\frac{\partial F}{\partial C_\nu^\mu} &= \frac{\partial}{\partial C_\nu^\mu} [\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, a^{[1]})] \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \frac{\partial}{\partial C_\nu^\mu} (C_i^j a^{[1]i} + c^j) \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \delta_\mu^j a^{[1]\nu} \\ &= \delta^{[2]}_\mu a^{[1]\nu} \\ &= [a^{[1]} \delta^{[2]T}]_\mu^\nu\end{aligned}$$

and hence that

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

Moreover, we also calculate

$$\frac{\partial F}{\partial c^\mu} = \sum_{j=1}^{m_2} \delta^{[2]j} \delta_\mu^j,$$

and hence that

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

Next we introduce another error term $\delta^{[1]} \in \mathbb{R}^{m_1}$ defined by

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

with coordinates

$$\begin{aligned} (\delta^{[1]\mu})^T &= \sum_{i=1}^{m_2} \sum_{j=1}^{m_1} \delta^{[2]i} C_j^i g^{[1]'}(z^{[1]j}) \delta_\mu^j \\ &= \sum_{i=1}^{m_2} \delta^{[2]i} C_\mu^i g^{[1]'}(z^{[1]\mu}) \end{aligned}$$

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

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

$$\begin{aligned} \frac{\partial F}{\partial B_\nu^\mu} &= \frac{\partial}{\partial B_\nu^\mu} [\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b))] \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{m_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^\lambda}{\partial B_\nu^\mu} \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \sum_{\lambda=1}^{m_1} \delta_\lambda^\rho g^{[1]'}(z^{[1]\rho}) \delta_\mu^\lambda x^\nu \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} \frac{\partial \Phi^j}{\partial \xi^\rho} \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} \sum_{\rho=1}^{m_1} C_\rho^j \delta_\mu^\rho g^{[1]'}(z^{[1]\rho}) x^\nu \\ &= \sum_{j=1}^{m_2} \delta^{[2]j} C_\mu^j g^{[1]'}(z^{[1]\mu}) x^\nu \\ &= \delta_\mu^{[1]} x^\nu \\ &= [x \delta^{[1]T}]_\mu^\nu, \end{aligned}$$

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. Let

$$S^i = x^i \circ \text{softmax}(z),$$

denote the i -th component of $\text{softmax}(z)$, and so we calculate

$$\begin{aligned} \frac{\partial S^i}{\partial z^j} &= \frac{\partial}{\partial z^j} \left[\left(\sum_{k=1}^m e^{z^k} \right)^{-1} e^{z^i} \right] \\ &= - \left(\sum_{k=1}^m e^{z^k} \right)^{-2} \left(\sum_{k=1}^m e^{z^k} \delta_j^k \right) e^{z^i} + \left(\sum_{k=1}^m e^{z^k} \right)^{-1} e^{z^i} \delta_j^i \\ &= - \left(\sum_{k=1}^m e^{z^k} \right)^{-2} e^{z^j} e^{z^i} + S^i \delta_j^i \\ &= -S^j S^i + S^i \delta_j^i \\ &= S^i (\delta_j^i - S^j). \end{aligned}$$

That is, as a map $dS_z : T_z \mathbb{R}^m \rightarrow T_{S(z)} \mathbb{R}^m$, we have that

$$dS_z = [S^i(\delta_j^i - S_j)]_j^i,$$

and we make note that dS_z is symmetric.

2.3 Binary Classification - An Example

We return the network given by

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \\ \vdots \\ z^{[2]} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]} \\ \vdots \\ a^{[2]} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

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 $m_2 = 1$ 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.

2.4 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:

$$\begin{array}{ccccccc}
 \underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} & \xrightarrow{\varphi^{[1]}} & \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} & \xrightarrow{g^{[1]}} & \underbrace{\begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} & \xrightarrow{\varphi^{[2]}} & \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} & \xrightarrow{g^{[2]}} & \underbrace{\begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} & \xrightarrow{\varphi^{[3]}} \dots \\
 & & & & & & & & & \\
 \dots \xrightarrow{\varphi^{[L-1]}} & & \underbrace{\begin{bmatrix} z^{[L-1]1} \\ \vdots \\ z^{[L-1]m_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} & \xrightarrow{g^{[L-1]}} & \underbrace{\begin{bmatrix} a^{[L-1]1} \\ \vdots \\ a^{[L-1]m_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} & \xrightarrow{\varphi^{[L]}} & \underbrace{\begin{bmatrix} z^{[L]1} \\ \vdots \\ z^{[L]m_L} \end{bmatrix}}_{\text{Layer } L} & \xrightarrow{g^{[L]}} & \underbrace{\begin{bmatrix} a^{[L]1} \\ \vdots \\ a^{[L]m_L} \end{bmatrix}}_{\text{Layer } L} & \xRightarrow{=} \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{m_L} \end{bmatrix},
 \end{array}$$

where

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

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

and

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

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 Backward Propagation

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 $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \rightarrow \mathbb{R}$ 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.

3.2 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             db1.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

```

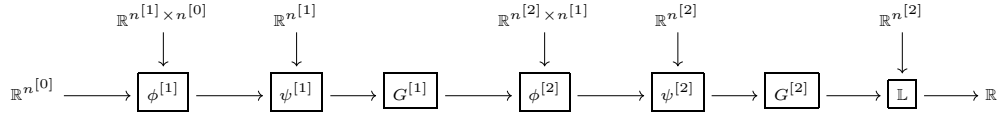
```

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}')

```

3.3 Better Backpropagation

We consider a neural network of the form



where we have the functions:

1.

$$G^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \rightarrow \mathbb{R}^{n^{[\ell]}}$$

is the broadcasting of the activation unit $g^{[\ell]} : \mathbb{R} \rightarrow \mathbb{R}$.

2.

$$\phi^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}} \times \mathbb{R}^{n^{[\ell-1]}} \rightarrow \mathbb{R}^{n^{[\ell]}}$$

is given by

$$\phi^{[\ell]}(W, x) = Wx.$$

3.

$$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}} \rightarrow \mathbb{R}^{n^{[\ell]}}$$

is given by

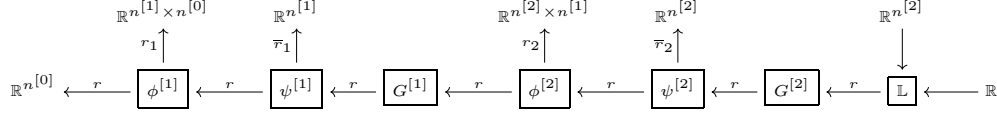
$$\psi^{[\ell]}(b, x) = x + b.$$

4.

$$\mathbb{L} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]}} \rightarrow \mathbb{R}$$

is the given loss-function.

We now consider back-propagating through the neural network via “reverse exterior differentiation”. We represent our various reverse derivatives via the following diagram:



First, we need to consider our individual derivatives:

1. Suppose $G : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is the broadcasting of $g : \mathbb{R} \rightarrow \mathbb{R}$. Then for $(x, \xi) \in T\mathbb{R}^n$, we have that

$$\begin{aligned} dG_x(\xi) &= G'(x) \odot \xi \\ &= \text{diag}(G'(x)) \cdot \xi \end{aligned}$$

and for any $\zeta \in T_{G(x)}\mathbb{R}^n$, the reverse derivative is given by

$$\begin{aligned} rG_x(\zeta) &= G'(x) \odot \zeta \\ &= \text{diag}(G'(x)) \cdot \zeta. \end{aligned}$$

2. Suppose $\phi : \mathbb{R}^{m \times n} \times \mathbb{R}^n \rightarrow \mathbb{R}^m$ is given by

$$\phi(A, x) = Ax.$$

Then we have two differentials to consider:

- (a) For any $(A, x) \in \mathbb{R}^{m \times n} \times \mathbb{R}^n$ and any $\xi \in T_x\mathbb{R}^n$, we have that

$$\begin{aligned} d\phi_{(A,x)}(\xi) &= A\xi \\ &= L_A(\xi); \end{aligned}$$

and for any $\zeta \in T_{\phi(A,x)}\mathbb{R}^m$, we have the reverse derivative

$$\begin{aligned} r\phi_{(A,x)}(\zeta) &= A^T \zeta \\ &= L_{A^T}(\zeta); \end{aligned}$$

where $L_A(B) = AB$, i.e., left-multiplication by A .

(b) For any $(A, x) \in \mathbb{R}^{m \times n} \times \mathbb{R}^n$ and any $Z \in T_A \mathbb{R}^{m \times n}$ we have that

$$\begin{aligned} d_1 \phi_{(A, x)}(Z) &= Zx \\ &= R_x(Z); \end{aligned}$$

and for any $\zeta \in T_{\phi(A, x)} \mathbb{R}^m$, we have the reverse derivative

$$\begin{aligned} r_1 \phi_{(A, x)}(\zeta) &= \zeta x^T \\ &= R_{x^T}(\zeta); \end{aligned}$$

where $R_A(B) = BA$, i.e, right-multiplication by A .

3. Suppose $\psi : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ is given by

$$\psi(b, x) = x + b.$$

Then we again have two (identical) differentials to consider:

(a) For any $(x, b) \in \mathbb{R}^n \times \mathbb{R}^n$ and any $\xi \in T_x \mathbb{R}^n$, we have that

$$d\psi_{(b, x)}(\xi) = \xi;$$

and for any $\zeta \in T_{\psi(b, x)} \mathbb{R}^n$, we have the reverse derivative

$$r\psi_{(b, x)}(\zeta) = \zeta.$$

(b) For any $(x, b) \in \mathbb{R}^n \times \mathbb{R}^n$ and any $\eta \in T_b \mathbb{R}^n$, we have that

$$d_1 \psi_{(b, x)}(\eta) = \eta;$$

and for any $\zeta \in T_{\psi(b, x)} \mathbb{R}^n$, we have the reverse derivative

$$\bar{r}_1 \psi_{(b, x)}(\zeta) = \zeta.$$

Proposition 3.1. *Suppose we have the compositional diagram*

$$\mathbb{R}^n \xrightarrow{f} \mathbb{R}^m \xrightarrow{g} \mathbb{R}^k \xrightarrow{h} \mathbb{R}^l$$

and we let $F = h \circ g \circ f : \mathbb{R}^n \rightarrow \mathbb{R}^l$. Then for any $x \in \mathbb{R}^n$ and any $\zeta \in T_{F(x)} \mathbb{R}^l$, the reverse derivative satisfies

$$rF_x(\zeta) = rf_x \circ rg_{f(x)} \circ rh_{g(f(x))}(\zeta).$$

Proof: For any $\xi \in T_x \mathbb{R}^n$ and any $\zeta \in T_{F(x)} \mathbb{R}^l$, we have by definition

$$\begin{aligned}
\langle rF_x(\zeta), \xi \rangle_{\mathbb{R}^n} &= \langle \zeta, dF_x(\xi) \rangle_{\mathbb{R}^l} \\
&= \langle \zeta, dh_{g(f(x))} \circ dg_{f(x)} \circ df_x(\xi) \rangle_{\mathbb{R}^l} \\
&= \langle rh_{g(f(x))}(\zeta), dg_{f(x)} \circ df_x(\xi) \rangle_{\mathbb{R}^k} \\
&= \langle rg_{f(x)} \circ rh_{g(f(x))}(\zeta), df_x(\xi) \rangle_{\mathbb{R}^m} \\
&= \langle rf_x \circ rg_{f(x)} \circ rh_{g(f(x))}(\zeta), \xi \rangle_{\mathbb{R}^n}
\end{aligned}$$

as desired. \square

Lemma 3.2. Suppose $f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^k$, and for $P \in \mathbb{R}^{n \times m}$, let $R = rf_P$. Then $R \in \mathbb{R}^{k \times n \times m}$ is rank $(1, 2)$ -tensor written in coordinates as

$$R = R_i^\mu{}_\nu \frac{\partial}{\partial X_\nu^\mu} \otimes dx^i,$$

and the components is given by

$$R_i^\mu{}_\nu = \frac{\partial f^i}{\partial X_\mu^\nu}$$

Proof: Considering the basis vectors $\frac{\partial}{\partial X_\mu^\nu} \in T_P \mathbb{R}^{n \times m}$ and $\frac{\partial}{\partial x^i} \in T_{f(P)} \mathbb{R}^k$ we have that

$$\begin{aligned}
R_i^\mu{}_\nu &= \left\langle R \left(\frac{\partial}{\partial x^i} \right), \frac{\partial}{\partial X_\mu^\nu} \right\rangle_F \\
&= \left\langle \frac{\partial}{\partial x^i}, df_P \left(\frac{\partial}{\partial X_\mu^\nu} \right) \right\rangle_{\mathbb{R}^k} \\
&= \left\langle \frac{\partial}{\partial x^i}, \frac{\partial f^\alpha}{\partial X_\mu^\nu} \frac{\partial}{\partial x^\alpha} \right\rangle_{\mathbb{R}^k} \\
&= \delta_{i\alpha} \frac{\partial f^\alpha}{\partial X_\mu^\nu},
\end{aligned}$$

as desired. \square

Returning to our neural network, for each point (x_j, y_j) in our training set, we first let

$$F_j := \mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]},$$

and we have our cost function

$$\mathbb{J} := \frac{1}{N} \sum_{j=1}^N F_j.$$

We use the following notation for our inputs and outputs of our respective functions:

•

$$\phi^{[\ell]} : (W^{[\ell]}, a^{[\ell-1]}_j) \mapsto u^{[\ell]}_j,$$

•

$$\psi^{[\ell]} : (b^{[\ell]}, u^{[\ell]}_j) \mapsto v^{[\ell]}_j,$$

•

$$G^{[\ell]} : v^{[\ell]}_j \mapsto a^{[\ell]}_j.$$

Let $p = (W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]})$ is a point in our parameter space. Suppose we wish to apply gradient descent with learning rate $\alpha \in T_{\mathbb{J}(p)}\mathbb{R}$, we would define our parameter updates via

$$W^{[1]} := W^{[1]} - r_1 \mathbb{J}_p(\alpha)$$

$$b^{[1]} := b^{[1]} - \bar{r}_1 \mathbb{J}_p(\alpha)$$

$$W^{[2]} := W^{[2]} - r_2 \mathbb{J}_p(\alpha)$$

$$b^{[2]} := b^{[2]} - \bar{r}_2 \mathbb{J}_p(\alpha).$$

Moreover, by linearity (and independence of our training data), we see that

$$r \mathbb{J}_p = \frac{1}{N} \sum_{j=1}^N r(F_j)_p,$$

so we need only calculate the various reverse derivatives of F_j .

To this end, we suppress the index j when we're working with the compositional function F . We calculate the reverse derivatives in the order traversed in our back-propagating path along the network.

1. $\bar{r}_2 \mathbb{J}_p$:

$$\begin{aligned} \bar{r}_2 F_p &= \bar{r}_2 (\mathbb{L} \circ G^{[2]} \circ \psi^{[2]})_p \\ &= \bar{r}_2 \psi_p^{[2]} \circ r G_{v^{[2]}}^{[2]} \circ r \mathbb{L}_{a^{[2]}} \\ &= \mathbb{1} \circ r G_{v^{[2]}}^{[2]} \circ r \mathbb{L}_{a^{[2]}} \\ &= r G_{v^{[2]}}^{[2]} \circ r \mathbb{L}_{a^{[2]}}, \end{aligned}$$

and hence

$$\bar{r}_2 \mathbb{J}_p = \frac{1}{N} \sum_{j=1}^N rG_{v^{[2]}_j}^{[2]} \cdot r\mathbb{L}_{a^{[2]}_j}$$

2. $r_2 \mathbb{J}_p$:

$$\begin{aligned} r_2 F_p &= r_2 (\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]})_p \\ &= r_2 \phi_p^{[2]} \circ r\psi_{u^{[2]}}^{[2]} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{a^{[1]}T} \circ \mathbf{1} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{a^{[1]}T} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}}, \end{aligned}$$

and hence

$$r_2 \mathbb{J}_p = \frac{1}{N} \sum_{j=1}^N R_{a^{[1]}T_j} \circ rG_{v^{[2]}_j}^{[2]} \cdot r\mathbb{L}_{a^{[2]}_j}.$$

Notice that this is not just a sum after matrix multiplication since we have composition with an operator, namely, $R_{a^{[1]}T_j}$. However, since the learning rate $\alpha \in T_{\mathbb{J}(p)}\mathbb{R} \cong \mathbb{R}$, which may pass through the aforementioned linear composition, we conclude that

$$\begin{aligned} r_2 \mathbb{J}_p &= \frac{1}{N} \sum_{j=1}^N R_{a^{[1]}T_j} \circ rG_{v^{[2]}_j}^{[2]} \cdot r\mathbb{L}_{a^{[2]}_j} \\ &= \frac{1}{N} \sum_{j=1}^N rG_{v^{[2]}_j}^{[2]} \cdot r\mathbb{L}_{a^{[2]}_j} a^{[1]T_j}. \end{aligned}$$

3. $\bar{r}_1 \mathbb{J}_p$:

$$\begin{aligned} \bar{r}_1 F_p &= \bar{r}_1 (\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]})_p \\ &= \bar{r}_1 \psi_p^{[1]} \circ rG_{v^{[1]}}^{[1]} \circ r\phi_{a^{[1]}}^{[2]} \circ r\psi_{u^{[2]}}^{[2]} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\ &= \mathbf{1} \circ rG_{v^{[1]}}^{[1]} \circ L_{W^{[2]}T} \circ \mathbf{1} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\ &= rG_{v^{[1]}}^{[1]} \circ L_{W^{[2]}T} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}}, \end{aligned}$$

and hence

$$\bar{r}_1 \mathbb{J}_p = \frac{1}{N} \sum_{j=1}^N rG_{v^{[1]}_j}^{[1]} \cdot W^{[2]T} \cdot rG_{v^{[2]}_j}^{[2]} \cdot r\mathbb{L}_{a^{[2]}_j}.$$

4. $r_1\mathbb{J}_p$:

$$\begin{aligned}
r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\
&= r_1\phi_p^{[1]} \circ r\psi_{u^{[1]}}^{[1]} \circ rG_{v^{[1]}}^{[1]} \circ r\phi_{a^{[1]}}^{[2]} \circ r\psi_{u^{[2]}}^{[2]} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\
&= R_{x^T} \circ \mathbb{1} \circ rG_{v^{[1]}}^{[1]} \circ L_{W^{[2]T}} \circ \mathbb{1} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}} \\
&= R_{x^T} \circ rG_{v^{[1]}}^{[1]} \circ L_{W^{[2]T}} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{a^{[2]}},
\end{aligned}$$

and hence

$$\begin{aligned}
r_1\mathbb{J}_p &= \frac{1}{N} \sum_{j=1}^N R_{x^T} \circ rG_{v^{[1]}}^{[1]} \cdot W^{[2]T} \cdot rG_{v^{[2]}}^{[2]} \cdot r\mathbb{L}_{a^{[2]}} \\
&= \frac{1}{N} \sum_{j=1}^N rG_{v^{[1]}}^{[1]} \cdot W^{[2]T} \cdot rG_{v^{[2]}}^{[2]} \cdot r\mathbb{L}_{a^{[2]}} \cdot x^T
\end{aligned}$$

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.

4.1 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 $\{(x_j, y_j)\} \subset \mathbb{R}^{m_0} \times \mathbb{R}^{m_L}$ with N examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \rightarrow \mathbb{R}$, 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.

5.1 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            w1.shape = (layers[1], layers[1-1])
15        params['b'][1] : array_like
16            b1.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

```

5.2 (Inverted) Dropout Regularization

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

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{g^{[1]}} \underbrace{\begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{g^{[2]}} \underbrace{\begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{\varphi^{[3]}} \text{output},$$

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

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

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.

5.2.1 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             wl.shape = (layers[l], layers[l-1])
42         params['b'][l] : array_like
43             bl.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

```

5.3 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).

5.4 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

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

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

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

6.2 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

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

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 $\{\mathbb{X}^k\}$.
 - b. For $1 \leq k \leq B$:
 - i. Perform forward propagation on \mathbb{X}^k :

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

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

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

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

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

iv. Perform gradient descent:

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

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.

7.6 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'][1].shape = w[1].shape
91         v['b'][1].shape = b[1].shape
92     grads : Dict[Dict]
93         grads['w'][1] : array_like
94             dw[1].shape = w[1].shape
95         grads['b'][1] : array_like
96             db[1].shape = b[1].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 us 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.

8.1 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

See [1].

We recall feature-normalization: Suppose $x \in \mathbb{R}^{m \times n}$ is some training data, and 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 (we shall not use it on the output 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.

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 (thus allowing for identity activation units), 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, let \hat{x} denote the

normalized x , and consider

$$\begin{aligned}
\mathbb{E}[\gamma \odot \hat{X} + \beta] &= \frac{1}{n} \sum_{j=1}^n (\gamma \odot \hat{x}_j + \beta) \\
&= \gamma \odot \mathbb{E}[\hat{X}] + \beta \\
&= 0 + \beta \\
&= \beta,
\end{aligned}$$

and so the new mean would be given by β . Similarly,

$$\begin{aligned}
\mathbb{E}[(\gamma \odot \hat{X} + \beta - \beta)^2] &= \frac{1}{n} \sum_{j=1}^n (\gamma \odot \hat{x}_j)^2 \\
&= \frac{1}{n} \sum_{j=1}^n (\gamma^2 \odot \hat{x}_j^2) \\
&= \gamma^2 \odot \mathbb{E}[(\hat{X} - 0)^2] \\
&= \gamma^2 \odot 1 \\
&= \gamma^2
\end{aligned}$$

and so we see the new variance would be given by γ^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 and focused on layer- ℓ , with 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 . However, we notice that

$$\begin{aligned} a^{[\ell-1]} &\mapsto \gamma^{[\ell]} \odot \frac{W^{[\ell]} a^{[\ell-1]} + b^{[\ell]} - \mu^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}} + \beta^{[\ell]} \\ &= \frac{\gamma^{[\ell]}}{\sqrt{\sigma^{[\ell]2} + \epsilon}} (W^{[\ell]} a^{[\ell-1]} - \mu^{[\ell]}) + \beta^{[\ell]}, \end{aligned}$$

after absorbing the $b^{[\ell]}$ into the parameter $\beta^{[\ell]}$. That is, we have 3 trainable parameters given by $W^{[\ell]} \in \mathbb{R}^{m_\ell \times m_{\ell-1}}$, $\gamma^{[\ell]}, \beta^{[\ell]} \in \mathbb{R}^{m_\ell}$.

9.1 Backward Propagation

We now show how batch normalization affects the backward propagation algorithm. For illustrative purposes, we assume a 2-layer neural network with arbitrary activation functions and generic loss function. We recall the setup (without bias $b^{[\ell]}$) used in [Section 2.1](#)

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\Phi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{BN_{\gamma^{[1]}, \beta^{[1]}}} \begin{bmatrix} \tilde{z}^{[1]1} \\ \vdots \\ \tilde{z}^{[1]m_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\Phi^{[2]}} \dots$$

$$\dots \xrightarrow{\Phi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]m_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]m_2} \end{bmatrix}}_{\text{Layer 2}} \Rightarrow \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{m_2} \end{bmatrix},$$

where

$$\Phi^{[1]} : \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \rightarrow \mathbb{R}^{m_1}, \quad \Phi^{[1]}(A, x) = Ax;$$

and

$$\Phi^{[2]} : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \rightarrow \mathbb{R}^{m_2}, \quad \Phi^{[2]}(A, b, x) = Ax + b.$$

Define the compositional function

$$G : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \rightarrow \mathbb{R},$$

given by

$$G(B, b, \gamma, \beta, A, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_{\gamma, \beta}(\Phi^{[1]}(A, x))).$$

This leads to compute some auxiliary differentials before continuing further.

Since we don't use batch normalization on the output layer, the bias term still exists.

Lemma 9.1. *For $N \in \mathbb{N}$, we define the expectation function $\mathbb{E} : \mathbb{R}^N \rightarrow \mathbb{R}$ given by*

$$\mathbb{E}[(x_1, \dots, x_N)] = \frac{1}{N} \sum_{j=1}^N x_j.$$

Let $z = \{z_1, \dots, z_N\} \subset \mathbb{R}$ be fixed, and define the mean

$$\mu := \mathbb{E}[z] = \frac{1}{N} \sum_{j=1}^N z_j.$$

Then as a differential, we have that $d\mathbb{E}_z : T_z \mathbb{R}^N \rightarrow T_\mu \mathbb{R}$ given by

$$d\mathbb{E}_z = \frac{1}{N} \sum_{j=1}^N dx_j|_{x=z}, \quad d\mathbb{E}_z(v) = \frac{1}{N} \sum_{j=1}^N v^j.$$

Moreover, for $\alpha = 1, \dots, N$, let $\iota_{z_\alpha} : \mathbb{R} \rightarrow \mathbb{R}^N$ denote the inclusion

$$\iota_{z_\alpha}(x) = (z_1, \dots, z_{\alpha-1}, x, z_{\alpha+1}, \dots, z_N).$$

Then the differentials

$$d_\alpha \mathbb{E}_{z_\alpha} := d(\mathbb{E} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha} \mathbb{R} \rightarrow T_\mu \mathbb{R},$$

are given by

$$\begin{aligned} d_\alpha \mathbb{E}_{z_\alpha} &= d(\mathbb{E} \circ \iota_{z_\alpha})_{z_\alpha} \\ &= d\mathbb{E}_z \cdot d(\iota_{z_\alpha})_{z_\alpha} \\ &= \frac{1}{N} dx_{z_\alpha}. \end{aligned}$$

Similarly, we define the variance function $\mathbb{V} : \mathbb{R}^N \rightarrow \mathbb{R}$ given by

$$\mathbb{V}[(x_1, \dots, x_N)] = \frac{1}{N} \sum_{j=1}^N (x_j - \mathbb{E}[(x_1, \dots, x_N)])^2.$$

For fixed z , define the variance

$$\sigma^2 = \mathbb{V}[z].$$

Then as a differential, we have that $d\mathbb{V}_z : T_z \mathbb{R}^N \rightarrow T_{\sigma^2} \mathbb{R}$ given by

$$d\mathbb{V}_z = \frac{2}{N} \sum_{j=1}^N (z_j - \mu) dx^j|_{x=z}, \quad d\mathbb{V}_z(v) = \frac{2}{N} \sum_{j=1}^N (z_j - \mu) v^j.$$

Moreover, for $\alpha = 1, \dots, N$, the differentials

$$d_\alpha \mathbb{V}_{z_\alpha} := d(\mathbb{V} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha} \mathbb{R} \rightarrow T_{\sigma^2} \mathbb{R}$$

are given by

$$\begin{aligned} d_\alpha \mathbb{V}_{z_\alpha} &= d(\mathbb{V} \circ \iota_{z_\alpha})_{z_\alpha} \\ &= d\mathbb{V}_z \cdot d(\iota_{z_\alpha})_{z_\alpha} \\ &= \frac{2}{N} (z_\alpha - \mu) dx_{z_\alpha} \end{aligned}$$

Proof: Immediate from direct calculation. \square

Corollary 9.2. For $\alpha = 1, \dots, N$, let $\mathcal{N}_\alpha : \mathbb{R}^{m \times N} \rightarrow \mathbb{R}^m$ denote the α -th component of the vector-valued, normalization transformation. That is,

$$\hat{x}_\alpha = \mathcal{N}_\alpha(x_1, \dots, x_N),$$

with

$$\hat{x}_\alpha^i = \frac{\pi_\alpha(x^i) - \mathbb{E}[x^i]}{(\mathbb{V}[x^i] + \epsilon)^{\frac{1}{2}}},$$

where $\pi_\alpha : \mathbb{R}^N \rightarrow \mathbb{R}$ is the projection onto the α -th coordinate

$$\pi_\alpha(x_1, \dots, x_N) = x_\alpha.$$

Fix $z_1, \dots, z_N \in \mathbb{R}^m$, let $\mu = \mathbb{E}[z] \in \mathbb{R}^m$ denote vector-mean and let $\sigma^2 = \mathbb{V}[z] \in \mathbb{R}^m$ denote the component-wise, vector-variation (i.e., $(\sigma^2)^i = \mathbb{V}[z^i]$). Then the differentials

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} := d(\mathcal{N}_\alpha \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha} \mathbb{R}^m \rightarrow T_{z_\alpha} \mathbb{R}^m$$

are given by the diagonal matrices

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} = \left(\frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{1}{N} \frac{(z_\alpha^i - \mu^i)^2}{((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) \delta_j^i.$$

Proof: We compute directly after noting that

$$d_\alpha(\mathcal{N}_\alpha)_{z_\alpha} = \begin{bmatrix} d_\alpha(\hat{x}_\alpha^1)_{z_\alpha^1} & \cdots & 0 \\ 0 & \ddots & 0 \\ 0 & \cdots & d_\alpha(\hat{x}_\alpha^m)_{z_\alpha^m} \end{bmatrix}$$

To this end, fix $1 \leq i \leq m$ and we compute

$$\begin{aligned} d_\alpha(\hat{x}_\alpha^i)_{z_\alpha^i} &= d_\alpha(\mathcal{N}_\alpha^i)_{z_\alpha^i} \\ &= \frac{d_\alpha(\pi_\alpha)_{z_\alpha^i} - d_\alpha \mathbb{E}_{z_\alpha^i}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{z_\alpha^i - \mu^i}{2((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} d_\alpha \mathbb{V}_{z_\alpha^i} \\ &= \frac{dx_{z_\alpha^i} - \frac{1}{N} dx_{z_\alpha^i}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{z_\alpha^i - \mu^i}{2((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \left(\frac{2}{N} (z_\alpha^i - \mu^i) dx_{z_\alpha^i} \right) \\ &= \left(\frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{(z_\alpha^i - \mu^i)^2}{N((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) dz_\alpha^i, \end{aligned}$$

as desired. \square

Proposition 9.3. Let $\mathcal{N} : \mathbb{R}^{m \times N} \rightarrow \mathbb{R}^{m \times N}$ denote the usual normalization transformation with $\hat{x}_\alpha = \mathcal{N}_\alpha(x)$. Let $BN : \mathbb{R}^m \times \mathbb{R}^m \times \mathbb{R}^{m \times N} \rightarrow \mathbb{R}^{m \times N}$ denote the batch normalization transformation $[x_j] \mapsto [\tilde{x}_j]$, i.e.,

$$\tilde{x}_j^i = \gamma^i \hat{x}_j^i + \beta^i,$$

where $x^i \in \mathbb{R}^N$. Moreover, given $\gamma, \beta \in \mathbb{R}^m$, for $\alpha \in \{1, \dots, N\}$, let

$$BN_\alpha^{\gamma, \beta} : \mathbb{R}^{m \times N} \rightarrow \mathbb{R}^m$$

denote

$$BN_\alpha^{\gamma, \beta}(x) = \gamma \odot \mathcal{N}_\alpha(x) + \beta.$$

Fix $z_1, \dots, z_N \in \mathbb{R}^m$, and let

$$\hat{z}_\alpha = \mathcal{N}_\alpha(z_1, \dots, z_N) \in \mathbb{R}^m, \quad \mu^i = \mathbb{E}[z^i] \in \mathbb{R}, \quad (\sigma^2)^i = \mathbb{V}[z^i] \in \mathbb{R}.$$

For $\alpha \in \{1, \dots, N\}$, $z \in \mathbb{R}^{m \times N}$ and for $\gamma, \beta \in \mathbb{R}^m$, we have the differentials:

- $d(BN_\alpha^{\beta, z})_\gamma : T_\gamma \mathbb{R}^m \rightarrow T_{\hat{z}} \mathbb{R}^m$, is given by

$$d(BN_\alpha^{\beta, z})_\gamma(v) = \hat{z}_\alpha \odot v, \quad \frac{\partial \tilde{z}_\alpha^i}{\partial \gamma^j} = \hat{z}_\alpha^i \delta_j^i.$$

- $d(BN_\alpha^{\gamma, z})_\beta : T_\beta \mathbb{R}^m \rightarrow T_{\hat{z}} \mathbb{R}^m$ is given by

$$d(BN_\alpha^{\gamma, z})_\beta(v) = v, \quad \frac{\partial \tilde{z}_\alpha^i}{\partial \beta^j} = \delta_j^i.$$

- $d(BN_\alpha^{\gamma, \beta})_{\hat{z}_\alpha} : T_{\hat{z}_\alpha} \mathbb{R}^m \rightarrow T_{\hat{z}} \mathbb{R}^m$ is given by

$$d(BN_\alpha^{\gamma, \beta})_{\hat{z}_\alpha}(v) = \gamma \odot v, \quad \frac{\partial \tilde{z}_\alpha^i}{\partial \hat{z}_\alpha^j} = \gamma^i \delta_j^i.$$

- $d_\alpha(BN_\alpha^{\gamma, \beta})_{z_\alpha} := d(BN_\alpha^{\gamma, \beta} \circ \iota_{z_\alpha})_{z_\alpha} : T_{z_\alpha} \mathbb{R}^m \rightarrow T_{\hat{z}_\alpha} \mathbb{R}^m$ is given by

$$d_\alpha(BN_\alpha^{\gamma, \beta})_{z_\alpha} = (\gamma \odot) d_\alpha(\mathcal{N}_\alpha)_{z_\alpha},$$

$$\frac{\partial \tilde{z}_\alpha^i}{\partial z_\alpha^j} = \gamma^i \left(\frac{1 - \frac{1}{N}}{\sqrt{(\sigma^2)^i + \epsilon}} - \frac{(z_\alpha^i - \mu^i)^2}{N((\sigma^2)^i + \epsilon)^{\frac{3}{2}}} \right) \delta_j^i$$

Proof: Follows immediately from the previous Corollary. □

We now return to considering the compositional function

$$G : \mathbb{R}^{m_2 \times m_1} \times \mathbb{R}^{m_2} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1} \times \mathbb{R}^{m_1 \times m_0} \times \mathbb{R}^{m_0} \rightarrow \mathbb{R},$$

given by

$$G(B, b, \gamma, \beta, A, x_\alpha) = \mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_\alpha^{\gamma, \beta}(\Phi^{[1]}(A, x))).$$

We compute (and since $\alpha \in \{1, \dots, N\}$ is fixed, we ignore implied summation for the moment)

•

$$\begin{aligned}
d_B G_B(V) &= d_B(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]})_B(V) \\
&= \left. \frac{d}{dt} \right|_{t=0} \mathbb{L}_y \circ g^{[2]}((B + tV)a^{[1]}_\alpha + b) \\
&= (\delta^{[2]}_\alpha{}^T)_\rho \left. \frac{d}{dt} \right|_{t=0} [(B^\rho_\lambda + tV^\rho_\lambda)a^{[1]\lambda}_\alpha + b^\rho] \\
&= (\delta^{[2]}_\alpha{}^T)_\rho V^\rho_\lambda a^{[1]\lambda}_\alpha \\
&= (a^{[1]}_\alpha \delta^{[2]}_\alpha{}^T)_\rho^\lambda V^\rho_\lambda,
\end{aligned}$$

and hence

$$d_B G_B = a^{[1]}_\alpha \delta^{[2]}_\alpha{}^T, \quad \frac{\partial G}{\partial B} = \delta^{[2]}_\alpha a^{[1]}_\alpha{}^T.$$

•

$$\begin{aligned}
d_b G_b(v) &= d_B(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]})_b(v) \\
&= (\delta^{[2]}_\alpha{}^T)_\rho \left. \frac{d}{dt} \right|_{t=0} [B^\rho_\lambda a^{[1]\lambda}_\alpha + (b^\rho + tv^\rho)] \\
&= \delta^{[2]}_\alpha{}^T v
\end{aligned}$$

yielding

$$d_b G_b = \delta^{[2]}_\alpha{}^T, \quad \frac{\partial G}{\partial b} = \delta^{[2]}_\alpha.$$

•

$$\begin{aligned}
d_\gamma G_\gamma(\xi) &= d_\gamma(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_\alpha^{\beta, z^{[1]}_\alpha}))_\gamma(\xi) \\
&= (\delta^{[2]}_\alpha{}^T) \cdot B \cdot dg^{[1]}_{\hat{z}^{[1]}_\alpha}(\hat{z}_\alpha \odot \xi) \\
&= (\delta^{[2]}_\alpha{}^T) \cdot B \cdot dg^{[1]}_{\hat{z}^{[1]}_\alpha} \text{diag}(\hat{z}^{[1]}_\alpha) \xi \\
&= \delta^{[1]}_\alpha{}^T \text{diag}(\hat{z}^{[1]}_\alpha) \xi,
\end{aligned}$$

and so

$$d_\gamma G_\gamma = \delta^{[1]}_\alpha{}^T \text{diag}(\hat{z}^{[1]}_\alpha), \quad \frac{\partial G}{\partial \gamma} = \text{diag}(\hat{z}^{[1]}_\alpha) \delta^{[1]}_\alpha.$$

•

$$\begin{aligned} d_\beta G_\beta(\eta) &= d_\beta(\mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]}(B, b, g^{[1]} \circ BN_\alpha^{\gamma, z^{[1]}_\alpha}))_\beta(\eta) \\ &= \delta^{[1]}_\alpha{}^T \eta, \end{aligned}$$

thus

$$d_\beta G_\beta = \delta^{[1]}_\alpha{}^T, \quad \frac{\partial G}{\partial \beta} = \delta^{[1]}_\alpha.$$

•

$$\begin{aligned} d_A G_A(V) &= \delta^{[1]}_\alpha{}^T \cdot d_\alpha(BN_\alpha^{\gamma, \beta})_{z^{[1]}_\alpha} d\Phi_A^{[1]}(V) \\ &= \delta^{[1]}_\alpha{}^T \text{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{z^{[1]}_\alpha} V x_\alpha, \end{aligned}$$

and hence

$$\begin{aligned} d_A G_A &= x_\alpha \delta^{[1]}_\alpha{}^T \text{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{z^{[1]}_\alpha}, \\ \frac{\partial G}{\partial A} &= \text{diag}(\gamma) d_\alpha(\mathcal{N}_\alpha)_{z^{[1]}_\alpha} \delta^{[1]}_\alpha x_\alpha{}^T. \end{aligned}$$

Finally, since

$$\mathbb{J}(W^{[2]}, b^{[2]}, \gamma, \beta, W^{[1]}) = \frac{1}{N} \sum_{\alpha=1}^N G(W^{[2]}, b^{[2]}, \gamma, \beta, W^{[1]}, x_\alpha),$$

we've described our desired gradients after summation.

9.2 Inferencing

We note that in our computation for forward propagation, that our normalization transforms change with out batches. This leads to ambiguity when predicting a label for a new example. One fix would be to average our means and variances over our batches. That is, suppose during our iteration process, we have training-batches of the form $\{\mathbb{X}^k : 1 \leq k \leq K\}$, where each \mathbb{X}^k has cardinality $|\mathbb{X}^k| = n$. Then for each hidden-layer $\ell \in \{1, \dots, L-1\}$, we obtain the means

$$\mu^{[\ell]}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} z^{[\ell]},$$

and the variances

$$\sigma^{2[\ell]}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} (z^{[\ell]} - \mu^{[\ell]}_k)^2.$$

That is, for each hidden-layer ℓ , we have the collection

$$\{\mu^{[\ell]}_k : 1 \leq k \leq K\}$$

from which we average again to obtain

$$\mu^{[\ell]} := \frac{1}{K} \sum_{k=1}^K \mu^{[\ell]}_k,$$

and the collection

$$\{\sigma^{2[\ell]}_k : 1 \leq k \leq K\},$$

from which we use the unbiased estimate

$$\sigma^{2[\ell]} := \frac{n}{n-1} \frac{1}{K} \sum_{k=1}^K \sigma^{2[\ell]}_k.$$

These quantities are what we use when computing the batch-normalization transforms of the hidden units for new examples.

9.3 Algorithm Outline

Suppose we have a training set \mathbb{X} with which we wish to train a binary classification via an L -layer neural network. Let $N = |\mathbb{X}|$ and let $n = 2^p$ be the batch size with $K = \lceil \frac{N}{n} \rceil$ batches per epoch. Then our algorithm would be as follows:

1. Set hyper-parameters. Initialize parameters.

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

a. Generate batches $\{\mathbb{X}^k : 1 \leq k \leq K\}$.

b. For $1 \leq k \leq K$:

i. Perform forward propagation on \mathbb{X}^k :

•

$$z^{[1]} = W^{[1]}x$$

• For $\ell \in \{1, \dots, L-1\}$:

—

$$z^{[\ell]} = W^{[\ell]}a^{[\ell-1]}$$

—

$$\mu^{[\ell]}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} z^{[\ell]}$$

—

$$\sigma^{2[\ell]}_k = \frac{1}{n} \sum_{x \in \mathbb{X}^k} (z^{[\ell]} - \mu^{[\ell]}_k)^2$$

—

$$\hat{z}^{[\ell]} = (\sigma^{2[\ell]}_k + \epsilon)^{-\frac{1}{2}} \odot (z^{[\ell]} - \mu^{[\ell]}_k)$$

—

$$\tilde{z}^{[\ell]} = \gamma^{[\ell]} \odot \hat{z}^{[\ell]} + \beta^{[\ell]}$$

—

$$a^{[\ell]} = g^{[\ell]}(\tilde{z}^{[\ell]})$$

•

$$z^{[L]} = W^{[L]} a^{[L-1]} + b$$

•

$$a^{[L]} = g^{[L]}(z^{[L]})$$

ii. Compute cost \mathbb{J} on \mathbb{X}^k .

iii. Apply backwards propagation on \mathbb{X}^k to obtain

$$\frac{\partial \mathbb{J}}{\partial W^{[\ell]}}, \quad \frac{\partial \mathbb{J}}{\partial b}, \quad \frac{\partial \mathbb{J}}{\partial \gamma^{[\ell]}}, \quad \frac{\partial \mathbb{J}}{\partial \beta^{[\ell]}}.$$

iv. Update parameters.

3. Compute

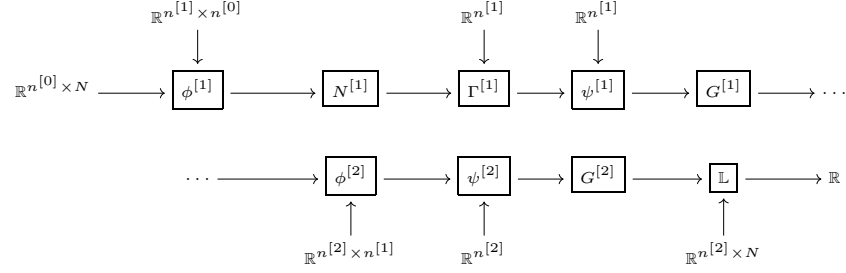
$$\begin{aligned} \mu^{[\ell]} &= \mathbb{E}[\mu^{[\ell]}_k], \\ \sigma^{2[\ell]} &= \frac{n}{n-1} \mathbb{E}[\sigma^{2[\ell]}_k] \end{aligned}$$

4. Return

$$W^{[\ell]}, \quad b, \quad \gamma^{[\ell]}, \quad \beta^{[\ell]}, \quad \mu^{[\ell]}, \quad \sigma^{2[\ell]}.$$

9.4 Better Backpropagation

We consider a neural network utilizing batch normalization of the form



where we have the functions

1.

$$G^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times N} \rightarrow \mathbb{R}^{n^{[\ell]} \times N}$$

is the broadcasting of the activation unit $g^{[\ell]} : \mathbb{R} \rightarrow \mathbb{R}$.

2.

$$\phi^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}} \times \mathbb{R}^{n^{[\ell-1]} \times N} \rightarrow \mathbb{R}^{n^{[\ell]} \times N}$$

is given by

$$\phi^{[\ell]}(W, x) = Wx.$$

3.

$$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]} \times N} \rightarrow \mathbb{R}^{n^{[\ell]} \times N}$$

is given by

$$\psi(b, x) = x + b\vec{1}^T,$$

where

$$\vec{1}^T = [1 \quad 1 \quad \dots \quad 1] \in \mathbb{R}^{n^{[\ell]}}.$$

4.

$$N^{[1]} : \mathbb{R}^{n^{[1]} \times N} \rightarrow \mathbb{R}^{n^{[1]} \times N}$$

is the normalization operator given by

$$N^{[1]} : x_j^i \mapsto \frac{x_j^i - \mathbb{E}[x^i]}{\sqrt{\mathbb{V}[x^i] + \epsilon}},$$

where \mathbb{E} is the expectation operator, i.e.,

$$\mathbb{E}[x^i] = \frac{1}{N} \sum_{j=1}^N x_j^i,$$

and \mathbb{V} is the variance operator, i.e.,

$$\mathbb{V}[x^i] = \mathbb{E}[(x^i - \mathbb{E}[x^i])^2].$$

5.

$$\Gamma^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]} \times N} \rightarrow \mathbb{R}^{n^{[\ell]} \times N}$$

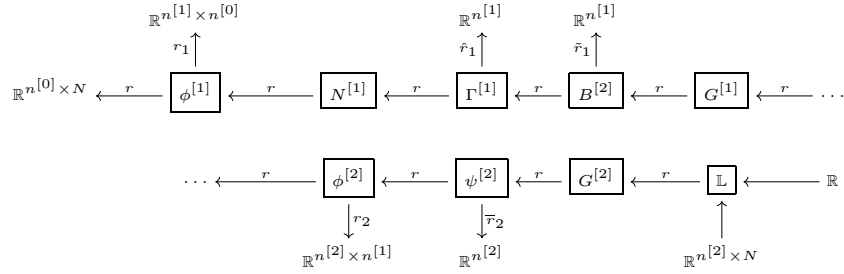
is given by

$$\Gamma(\gamma, x) = \gamma \vec{1}^T \odot x,$$

where

$$\vec{1}^T = [1 \ 1 \ \dots \ 1] \in \mathbb{R}^{n^{[\ell]}}.$$

We now consider back-propagating through the network via reverse differentiations as in the following diagram:



We now need to consider our individual derivatives:

1. Suppose $G : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$ is the broadcasting of $g : \mathbb{R} \rightarrow \mathbb{R}$. Then for any $(x, \xi) \in T\mathbb{R}^{m \times n}$ we have that

$$dG_x(\xi) = G'(x) \odot \xi.$$

Then for any $\zeta \in T_{G(x)}\mathbb{R}^{m \times n}$, we have the reverse derivative is given by

$$rG_x(\zeta) = G'(x) \odot \zeta.$$

2. Suppose $\phi : \mathbb{R}^{m \times n} \times \mathbb{R}^{n \times N} \rightarrow \mathbb{R}^{m \times N}$ is given by

$$\phi(W, x) = Wx.$$

Then we have two differential paths to consider:

- (a) For any $(W, x) \in \mathbb{R}^{m \times n} \times \mathbb{R}^{n \times N}$ and any $\xi \in T_x \mathbb{R}^{n \times N}$, we have that

$$\begin{aligned} d\phi_{(W,x)}(\xi) &= W \cdot \xi \\ &= L_W(\xi), \end{aligned}$$

and for any $\zeta \in T_{\phi(W,x)} \mathbb{R}^{m \times N}$, we have the reverse differential

$$\begin{aligned} r\phi_{(W,x)}(\zeta) &= W^T \cdot \zeta \\ &= L_{W^T}(\zeta). \end{aligned}$$

- (b) For any $(W, x) \in \mathbb{R}^{m \times n} \times \mathbb{R}^{n \times N}$ and any $Z \in T_W \mathbb{R}^{m \times n}$, we have that

$$\begin{aligned} d_1\phi_{(W,x)}(Z) &= Z \cdot x \\ &= R_x(Z), \end{aligned}$$

and for any $\zeta \in T_{\phi(W,x)} \mathbb{R}^{m \times N}$, we have the reverse differential

$$\begin{aligned} r_1\phi_{(W,x)}(\zeta) &= \zeta \cdot x^T \\ &= R_{x^T}(\zeta). \end{aligned}$$

3. Suppose $\psi : \mathbb{R}^n \times \mathbb{R}^{n \times N} \rightarrow \mathbb{R}^{n \times N}$ is given by

$$\psi(b, x) = x + b\vec{1}^T,$$

where

$$\vec{1}^T = [1 \quad 1 \quad \dots \quad 1] \in \mathbb{R}^N.$$

Then we look at the two differential paths and for any $(b, x) \in \mathbb{R}^n \times \mathbb{R}^{n \times N}$ any $\xi \in T_x \mathbb{R}^{n \times N}$, $\eta \in T_b \mathbb{R}^n$ and $\zeta \in T_{\psi(b,x)} \mathbb{R}^{n \times N}$:

- (a) In the network direction, we have that

$$d\psi_{(b,x)}(\xi) = \xi,$$

with reverse differential

$$r\psi_{(b,x)}(\zeta) = \zeta.$$

(b) In the parameter-space direction, we have that

$$\begin{aligned}\bar{d}\psi_{(b,x)}(\eta) &= \eta \cdot \vec{1}^T \\ &= R_{\vec{1}^T}(\eta),\end{aligned}$$

with reverse differential

$$\begin{aligned}\bar{r}\psi_{(b,x)}(\zeta) &= \zeta \cdot \vec{1} \\ &= R_{\vec{1}}(\zeta).\end{aligned}$$

4. Suppose $\Gamma : \mathbb{R}^n \times \mathbb{R}^{n \times N} \rightarrow \mathbb{R}^{n \times N}$ is given by

$$\Gamma(\gamma, x) = \gamma \vec{1}^T \odot x.$$

The considering the two paths of differentiation, we have that for any $((\gamma, x), (\eta, \xi)) \in T(\mathbb{R}^n \times \mathbb{R}^{n \times N})$ and $\zeta \in T_{\Gamma(\gamma, x)}\mathbb{R}^{n \times N}$ that:

(a) In the network direction, we have that

$$d\Gamma_{(\gamma, x)}(\xi) = \gamma \vec{1}^T \odot \xi,$$

with reverse differential

$$r\Gamma_{(\gamma, x)}(\zeta) = \gamma \vec{1}^T \odot \zeta.$$

(b) In the parameter-space direction, we have that

$$\begin{aligned}\hat{d}\Gamma_{(\gamma, x)}(\eta) &= \eta \vec{1}^T \odot x \\ &= \odot_x \circ R_{\vec{1}^T}(\eta),\end{aligned}$$

with reverse differential

$$\begin{aligned}\hat{r}\Gamma_{(\gamma, x)}(\zeta) &= (x \odot \zeta) \cdot \vec{1} \\ &= R_{\vec{1}} \circ \odot_x(\zeta).\end{aligned}$$

5. As the normalization operator is quite involved, we move its computation to the following section.

10 Multi-Class Softmax Regression

Thus far, we've mostly been dealing with binary classification problems, that is, our true label y takes values in $\{0, 1\}$, where $y = 1$ represents when the object in question represents our desired classification, and $y = 0$ when it does not. However, in many examples we wish to expand upon this, for example, instead of knowing whenever an image contains a cat ($y = 1$) or it doesn't contain a cat ($y = 0$), maybe we would like to have a table of the following

Table 1: Classification

y	Label
$y = 0$	None of the following
$y = 1$	Cat
$y = 2$	Dog
$y = 3$	Bird
$y = 4$	Elephant
$y = 5$	Bear

That is, we have a total of 6 classes we wish to distinguish. If we were to train a neural network for this classification problem, the only time this needs to be considered is on the output layer. With this in mind, we shall only consider the simple regression problem

$$\begin{bmatrix} x^1 \\ \vdots \\ x^m \end{bmatrix} \xrightarrow{Wx+b} \begin{bmatrix} z^1 \\ \vdots \\ z^C \end{bmatrix} \xrightarrow{g(z)} \begin{bmatrix} a^1 \\ \vdots \\ a^C \end{bmatrix} \longrightarrow \hat{y},$$

where C is the number of labels in our classification.

First, we need to *one-hot encode* our labels. That is, if our labels are given by

$$\{0, 1, \dots, C-1\},$$

then we consider the basis vectors in \mathbb{R}^C

$$\{e_1, \dots, e_C\},$$

which clearly admits a bijection

$$\{0, 1, \dots, C-1\} \xrightarrow{\cong} \{e_1, \dots, e_C\}, \quad i \mapsto e_{i+1}.$$

Thus, we've effectively mapped our true labels

$$y \in \{0, 1, \dots, C-1\}^N \mapsto y \in \mathbb{R}^{C \times N},$$

where

$$(y = i) \mapsto (y = e_{i+1}).$$

Next, we need to decide which type of nonlinearity $g : \mathbb{R}^C \rightarrow \mathbb{R}^C$ to impose. To this end, we would like a^i to satisfy

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

then we can declare a prediction via

$$i_0 = \arg \max_i a^i, \quad \hat{y} = e_{i_0} \leftrightarrow \hat{y} = i_0 - 1.$$

That is, we would like our target output vector $a \in \mathbb{R}^C$ to be a probability distribution, i.e.,

$$0 \leq a^i \leq 1, i \in \{1, \dots, C\},$$

and

$$\sum_{i=1}^C a^i = 1.$$

This leads us to letting g be the softmax function, i.e.,

$$g(z^1, \dots, z^C) = \frac{1}{\sum_{i=1}^C e^{z^i}} \begin{bmatrix} e^{z^1} \\ \vdots \\ e^{z^C} \end{bmatrix}.$$

Finally, we need to define a cost function $\mathbb{L} : \mathbb{R}^C \times \mathbb{R}^C \rightarrow \mathbb{R}$ with which we can compare our true value to our predicted value. To this end, we consider the cross-entropy function \mathbb{L} defined by

$$\mathbb{L}(a_j, y_j) = - \sum_{i=1}^C y_j^i \log a_j^i.$$

We note that since $y_j = e_k$ for some $k \in \{1, \dots, C\}$, that this sum is actually a single element. Moreover, when $C = 2$, we recover our log-loss function for the sigmoid activation. This finally yields a cost function

$$\begin{aligned} \mathbb{J}(W, b) &= -\frac{1}{N} \sum_{j=1}^N \sum_{i=1}^C y_j^i \log a_j^i \\ &= -\frac{1}{N} (y : \log a), \end{aligned}$$

where

$$A : B = \langle A, B \rangle_F = \text{tr}(A^T B),$$

is the Frobenius norm on $\mathbb{R}^{C \times N}$.

To minimize our cost, we first note

$$\begin{aligned} \frac{\partial \mathbb{L}_y \circ g}{\partial z^\mu} &= \sum_{i=1}^C \frac{\partial \mathbb{L}_y}{\partial a^i} \frac{\partial S^i}{\partial z^\mu} \\ &= - \sum_{i=1}^C \frac{y^i}{a^i} a^i (\delta_\mu^i - a^\mu) \\ &= - \sum_{i=1}^C y^i (\delta_\mu^i - a^\mu) \\ &= -y^\mu + a^\mu \underbrace{\sum_{i=1}^C y^i}_{=1} \\ &= a^\mu - y^\mu, \end{aligned}$$

then we see that

$$\begin{aligned} \frac{\partial z^\mu}{\partial W_\beta^\alpha} &= \frac{\partial}{\partial W_\beta^\alpha} (W_k^\mu x^k + b^\mu) \\ &= \sum_{k=1}^m \delta_\alpha^\mu \delta_k^\beta x^k \\ &= \delta_\alpha^\mu x^\beta, \end{aligned}$$

and

$$\frac{\partial z^\mu}{\partial b^\alpha} = \delta_\alpha^\mu.$$

Hence,

$$\begin{aligned} \frac{\partial \mathbb{L}_y}{\partial W_\beta^\alpha} &= \sum_{\mu=1}^C (a^\mu - y^\mu) \delta_\alpha^\mu x^\beta \\ &= x(a - y)^T, \end{aligned}$$

yielding a gradient of

$$\frac{\partial \mathbb{L}_y}{\partial W} = (a - y)x^T,$$

and similarly

$$\begin{aligned}\frac{\partial \mathbb{L}_y}{\partial b^\alpha} &= \sum_{\mu=1}^C (a^\mu - y^\mu) \delta_\alpha^\mu \\ &= a^\alpha - y^\alpha,\end{aligned}$$

and so

$$\frac{\partial \mathbb{L}_y}{\partial b} = a - y.$$

Finally, we conclude that

$$\frac{\partial \mathbb{J}}{\partial W} = \frac{1}{N} \sum_{j=1}^N (a_j - y_j) (x_j)^T = \frac{1}{N} (a - y) x^T,$$

and

$$\frac{\partial \mathbb{J}}{\partial b} = \frac{1}{N} \sum_{j=1}^N (a_j - y_j).$$

We remark that for a deep neural network, the backwards propagation follows a similar path backwards through the network since we have the aforementioned differentials.

Part III

Convolutional Neural Networks

11 An Introduction to Convolutions

One common application of neural networks is that of image detection/-classification. Recall that an image in grayscale can be seen as a matrix $x \in \mathbb{R}^{m \times n}$, where

$$x_j^i \in \{0, 1, \dots, 9, 10\},$$

and 10 represents “white” and 0 represents “black”.

Instead of flattening the pixels into a vector $\vec{x} \in \mathbb{R}^{nm}$ and feeding the input into a deep network, we observe that several simple detections may be imposed on the image first while it’s in matrix form. That is, suppose we wish to detect vertical or horizontal edges in the image first. As there are typically several of such edges in an image, and these edges are the “atomic” pieces of full images, this initial detection would be of great benefit.

To this end, we wish to impose an operation which finds where a pixel x_j^i changes dramatically when moving to a neighboring pixel. One way to find these changes is with convolutions, or cross-correlations.

11.1 Cross-Correlation

We first recall that given two function $f, g : \mathbb{Z} \rightarrow \mathbb{R}$, the (discrete) cross-correlation $f * g$ is defined by

$$f * g(n) = \sum_{j=-\infty}^{\infty} f(j)g(j+n).$$

We note that cross-correlation is not commutative, however, we see that

$$\begin{aligned} g * f(-n) &= \sum_{j=-\infty}^{\infty} g(j)f(j-n) & i = j - n \\ &= \sum_{i=-\infty}^{\infty} f(i)g(i+n) \\ &= f * g(n). \end{aligned}$$

We may similarly define for $f, g : \mathbb{Z}^2 \rightarrow \mathbb{R}$,

$$f * g(k, l) = \sum_{(i,j) \in \mathbb{Z}^2} f(i, j)g(i+k, j+l).$$

Whenever f or g has finite support, say in $[-M, M]$, the above sum reduces to

$$f * g(n) = \sum_{j=-M}^M f(j)g(j+n).$$

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]}}$ and let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ with $f^{[1]} \leq \min\{n_h^{[0]}, n_w^{[0]}\}$. Define

$$n_\alpha^{[1]} = n_\alpha^{[0]} - f^{[1]} + 1, \quad \alpha = h, w,$$

and we obtain the matrix $F * x \in \mathbb{R}^{n_h^{[1]} \times n_w^{[1]}}$ given by

$$(F * x)_l^k = \sum_{i,j=1}^{f^{[1]}} F_j^i x_{j+l-1}^{i+k-1}.$$

Note that this is exactly the cross-correlation defined above, except with finite support and reindexed to start at 1.

In what follows, this cross-correlation operator will be called the *convolution* operator, and F will be called the filter (or kernel).

Example 11.1. *Suppose*

$$x = \begin{bmatrix} 1 & 2 & 0 & 3 \\ 4 & 5 & 6 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix}$$

and

$$F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Then $f = 2$, $n_h^{[0]} = 3$, $n_w^{[0]} = 4$, and so

$$n_h^{[1]} = 3 - 2 + 1 = 2,$$

$$n_w^{[1]} = 4 - 2 + 1 = 3.$$

We now compute $(F * x) \in \mathbb{R}^{2 \times 3}$

$$\begin{aligned} (F * x)_1^1 &= 1 * 1 + 0 * 2 + 1 * 4 + 1 * 5 = 10 \\ (F * x)_2^1 &= 1 * 2 + 0 * 0 + 1 * 5 + 1 * 6 = 13 \\ (F * x)_3^1 &= 1 * 0 + 0 * 3 + 1 * 6 + 1 * 0 = 6 \\ (F * x)_1^2 &= 1 * 4 + 0 * 5 + 1 * 0 + 1 * 1 = 5 \\ (F * x)_2^2 &= 1 * 5 + 0 * 6 + 1 * 1 + 2 * 2 = 10 \\ (F * x)_3^2 &= 1 * 6 + 0 * 0 + 1 * 2 + 1 * 3 = 11, \end{aligned}$$

and hence

$$F * x = \begin{bmatrix} 10 & 13 & 6 \\ 5 & 10 & 11 \end{bmatrix}.$$

Example 11.2. Suppose

$$x = \begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix},$$

which can be seen as a grayscale image that's white on the left half of the image and black on the right half. Now define the filter

$$F = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

Then $F * x \in \mathbb{R}^{4 \times 4}$ and is given by

$$F * x = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix},$$

which looks like an image a “white” edge in the middle, telling us the original has an edge in the middle that goes from “bright” pixels to “dark” pixels.

This idea of convolution seems to be able to detect our edges. However, we see that the pixels in the “interior” of the matrix affect the convolution much more than the pixels on the “boundary”. This may not always matter, but when it does, we need a technique to allow for the boundary pixels to be more prominent. One such fix is to add some “padding” around the original image.

11.2 Convolution with Padding

Suppose $x \in \mathbb{R}^{m \times n}$ is matrix, and let $p \in \mathbb{Z}_{\geq 0}$, which we will call the *padding*. Define a new matrix $(x, p) \in \mathbb{R}^{(m+2p) \times (n+2p)}$ given by

$$(x, p)_l^k = \begin{cases} x_{l-p}^{k-p} & \text{if } p < k \leq m + p \text{ and } p < l \leq n + p, \\ 0 & \text{else.} \end{cases}$$

Example 11.3. *Suppose*

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Then $(x, 0) = x$ immediately,

$$(x, 1) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},$$

$$(x, 2) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = ((x, 1), 1).$$

From the previous example, we see a recursive property with padding, i.e.,

$$\begin{aligned} (x, p) &= ((x, p-1), 1) \\ &= (((x, p-2), 1), 1) \\ &\vdots \\ &= (\underbrace{(\dots((x, 1), 1), \dots 1)}_{p\text{-times}}, 1) \end{aligned}$$

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter, and let $p \in \mathbb{Z}_{\geq 0}$ be the padding. Then since (x, p) is an $(n_h^{[0]} + 2p) \times (n_w^{[0]} + 2p)$ -matrix, we have that the convolution $F * (x, p)$ has a size given by

$$n_\alpha^{[1]} = n_\alpha^{[0]} + 2p - f^{[1]} + 1, \quad \alpha = h, w,$$

and we write

$$F *^p x = F * (x, p).$$

When $p = 0$, we say that $F *^p x$ is a *valid convolution*, and we'll typically drop the p -superscript. When $p = \frac{f^{[1]}-1}{2}$, we say that $F *^p x$ is a *same convolution*, since

$$n_\alpha^{[1]} = n_\alpha^{[0]}, \quad \alpha = h, w.$$

We remark here that in many application our desired filters have $f^{[1]}$ being odd (if it's not odd, then it cannot be a same convolution).

11.3 Strided Convolution

We note that in our definition of a convolution

$$(F * x)_l^k = \sum_{i,j=1}^{f^{[1]}} F_j^i x_{j+l-1}^{i+k-1},$$

that we're sliding our filter F along x with a *stride* of $s = 1$. This does not necessarily have to be the case. We modify our definition of convolution to allow for $s \in \mathbb{N}$ as follows:

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter and let $s \in \mathbb{N}$ be the stride. Let

$$n_\alpha^{[1]} = \lfloor \frac{n_\alpha^{[0]} - f^{[1]}}{s} + 1 \rfloor, \quad \alpha = h, w,$$

and define $F *_s x \in \mathbb{R}^{n_h^{[1]} \times n_w^{[1]}}$ to be the matrix given by

$$(F *_s x)_l^k = \sum_{i,j=1}^{f^{[1]}} F_j^i x_{j+s(l-1)}^{i+s(k-1)}.$$

We note that the definition of a strided convolution is a direct generalization of our previous definition of convolution, namely with stride $s = 1$.

Example 11.4. *Suppose*

$$x = \begin{bmatrix} 1 & 0 & 2 & 0 \\ 3 & 0 & 4 & 0 \\ 0 & 5 & 0 & 6 \\ 7 & 0 & 8 & 0 \end{bmatrix},$$

$$F = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix},$$

and suppose we have a stride of 2 (any larger stride would result in a (1×1) -matrix). Then we see that

$$n_\alpha^{[1]} = \lfloor \frac{4 - 2}{2} + 1 \rfloor = 2, \quad \alpha = h, w,$$

and hence that

$$\begin{aligned} (F *_2 x)_1^1 &= 1 * 1 + 1 * 0 + 2 * 3 + 0 * 0 = 7 \\ (F *_2 x)_2^1 &= 1 * 2 + 1 * 0 + 2 * 4 + 0 * 0 = 10 \\ (F *_2 x)_1^2 &= 1 * 0 + 1 * 5 + 2 * 7 + 0 * 0 = 19 \\ (F *_2 x)_2^2 &= 1 * 0 + 1 * 6 + 2 * 8 + 0 * 0 = 22, \end{aligned}$$

or rather

$$F *_2 x = \begin{bmatrix} 7 & 10 \\ 19 & 22 \end{bmatrix}.$$

11.4 Strided Convolutions with Padding

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]}}$, let $F \in \mathbb{R}^{f^{[1]} \times f^{[1]}}$ be a filter, let $s \in \mathbb{N}$ be the stride, and let $p \in \mathbb{Z}_{\geq 0}$ be the padding. We define

$$F *_s^p x := F *_s (x, p),$$

that is, we first pad x , then compute the strided convolution of the filter F with (x, p) . From our previous work, we see that for $\alpha = h, w$, that

$$\begin{aligned} n_\alpha^{[1]} &= \left\lfloor \frac{n_\alpha'^{[0]} - f^{[1]}}{s} + 1 \right\rfloor, \quad n' \sim (x, p) \\ &= \left\lfloor \frac{n_\alpha^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor. \end{aligned}$$

Moreover, to compute a closed form of the strided convolution with padding, we first define the set

$$\begin{aligned} \mathcal{I}_l^{[1]k} &= \mathcal{I}(n_h^{[0]}, n_w^{[0]}, p, s; k, l) \\ &:= \{(i, j) \in \mathbb{Z}^2 : p < i + s(k-1) - p \leq n_h^{[0]} + p ; \\ &\quad p < j + s(l-1) - p \leq n_w^{[0]} + p\} \\ &= \{(i, j) \in \mathbb{Z}^2 : 2p - s(k-1) < i \leq 2p - s(k-1) + n_h^{[0]} ; \\ &\quad 2p - s(l-1) < j \leq 2p - s(l-1) + n_w^{[0]}\} \end{aligned}$$

and now we immediately see by chasing the definitions that

$$\begin{aligned} (F *_s^p x)_l^k &= (F *_s (x, p))_l^k \\ &= \sum_{i,j=1}^{f^{[1]}} F_j^i(x, p)_{j+s(l-1)}^{i+s(k-1)} \\ &= \sum_{i,j=1}^{f^{[1]}} F_j^i x_{j+s(l-1)-p}^{i+s(k-1)-p} \chi_{\mathcal{I}_l^{[1]k}}(i, j) \end{aligned}$$

Example 11.5. *Suppose*

$$x = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 3 & 0 \\ 4 & 0 & 5 \end{bmatrix},$$

and we have a filter

$$F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

*We first compute $F *_{\frac{1}{2}} x$: Since we we're using a padding of $p = 1$, we have that*

$$(x, 1) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 4 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Using a stride of $s = 2$, we see we have resultant dimensions of the form

$$\begin{aligned} n_{\alpha}^{[1]} &= \lfloor \frac{3 + 2 * 1 - 2}{2} + 1 \rfloor \\ &= 2, \end{aligned}$$

*that is, $F *_{\frac{1}{2}} x \in \mathbb{R}^{2 \times 2}$. We now compute*

$$\begin{aligned} (F *_{\frac{1}{2}} x)_1^1 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 1 = 1 \\ (F *_{\frac{1}{2}} x)_2^1 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 2 = 2 \\ (F *_{\frac{1}{2}} x)_1^2 &= 1 * 0 + 1 * 0 + 0 * 0 + 1 * 4 = 4 \\ (F *_{\frac{1}{2}} x)_2^2 &= 1 * 0 + 1 * 0 + 0 * 5 + 1 * 0 = 0, \end{aligned}$$

or rather

$$F *_{\frac{1}{2}} x = \begin{bmatrix} 1 & 2 \\ 4 & 0 \end{bmatrix}.$$

11.5 Convolutions Over Volumes

At the beginning of this section, we began by considering a grayscale image which we represented as a matrix $x \in \mathbb{R}^{n_h \times n_w}$. Suppose that instead of grayscale, we have an RGB image. Then for each fixed color component, we may represent the component as a matrix as before. However, since flattening a color image into a grayscale image would break our desired

symmetries (e.g., for edges, etc), we would like a way to handle convolutions of an RGB image being represented as a rank-3 tensor $x \in \mathbb{R}^{n_h \times n_w \times n_c}$. This n_c parameter represents the “depth” of the image, which we shall call the *channels*. That is, x has a red, a green, and a blue channel. We wish to work with channels simultaneously to see simplifications in their relationships with each other. To this end, we introduce a notion of convolution over volumes, which instead of moving a $f^{[1]} \times f^{[1]}$ -square across x , we move a $f^{[1]} \times f^{[1]} \times n_c^{[0]}$ -prism across x instead.

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]} \times n_c^{[0]}}$, and suppose $F \in \mathbb{R}^{f^{[1]} \times f^{[1]} \times n_c^{[0]}}$ is a filter (noted the channel size of the input must match the channel size of the filter). Then as before we have that

$$n_\alpha^{[1]} = n_\alpha^{[0]} - f + 1, \quad \alpha = h, w,$$

and we define $F * x \in \mathbb{R}^{n_h^{[1]} \times n_w^{[1]}}$ by

$$(F * x)_l^k = \sum_{i,j=1}^{f^{[1]}} \sum_{\rho=1}^{n_c^{[0]}} F_{j,\rho}^i x_{j+l-1}^{i+k-1,\rho}.$$

Similarly, if $p \in \mathbb{Z}_{\geq 0}$ is the padding and $s \in \mathbb{N}$ is the stride, we have that

$$n_\alpha^{[1]} = \left\lfloor \frac{n_\alpha^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor, \quad \alpha = h, w,$$

and we define $F *_s^p x \in \mathbb{R}^{n_h^{[1]} \times n_w^{[1]}}$ by

$$(F *_s^p x)_l^k = \sum_{\rho=1}^{n_c^{[0]}} \sum_{i,j=1}^{f^{[1]}} F_{j,\rho}^i x_{j+s(l-1)-p}^{i+s(k-1)-p,\rho} \chi_{\mathcal{I}_l^{[1]}k}(i,j).$$

11.6 Multiple Filters

Suppose $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]} \times n_c^{[0]}}$, and we wish to convolve x with $n_c^{[1]}$ -filters, i.e.,

$$F_\eta \in \mathbb{R}^{f^{[1]} \times f^{[1]} \times n_c^{[0]}}, \quad \eta \in \{1, \dots, n_c^{[1]}\}.$$

Then we have that

$$n_\alpha^{[1]} = \left\lfloor \frac{n_\alpha^{[0]} + 2p - f^{[1]}}{s} + 1 \right\rfloor, \quad \alpha = h, w,$$

and letting $F = \{F_\eta : 1 \leq \eta \leq n_c^{[1]}\}$, we define $F *_s^p x \in \mathbb{R}^{n_h^{[1]} \times n_w^{[1]} \times n_c^{[1]}}$ to be given by

$$(F *_s^p x)_\eta^k = (F_\eta *_s^p x)_l^k.$$

12 Convolutional Networks

We've now seen enough of how to compute convolutions, and are ready to implement them into a neural network architecture. There are three main types of layers that occur in a convolutional neural network, namely, a convolutional layer (**conv**), a pooling layer (**pool**), and a fully connected layer (FC), which is the usual type of neural network layer we've seen previously.

12.1 Convolutional Layers (**conv**)

Suppose we are propagating from layer- ℓ to layer- $(\ell+1)$ in a neural network, and suppose $a^{[\ell]} \in \mathbb{R}^{n_h^{[\ell]} \times n_w^{[\ell]} \times n_c^{[\ell]}}$. Suppose we have $n_c^{[\ell+1]}$ -filters we wish to convolve with, each of size $f^{[\ell+1]} \times f^{[\ell+1]} \times n_c^{[\ell]}$, and we have padding $p^{[\ell+1]}$ and a stride $s^{[\ell+1]}$. We let $\text{conv}^{[\ell+1]}(a^{[\ell]})$ denote the mapping:

- For $\eta \in \{1, \dots, n_c^{[\ell+1]}\}$, compute

$$F_\eta *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + b_\eta^{[\ell+1]},$$

where $b_\eta^{[\ell+1]} \in \mathbb{R}$ and the sum is a broadcasting.

- Stack the resultant matrices to obtain an $n_h^{[\ell+1]} \times n_w^{[\ell+1]} \times n_c^{[\ell+1]}$ -tensor.

$$\text{conv}^{[\ell+1]}(a^{[\ell]}) = F *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + b^{[\ell+1]}$$

Letting

$$z^{[\ell+1]} = \text{conv}^{[\ell+1]}(a^{[\ell]}),$$

we may then apply our activation unit for the layer $g^{[\ell+1]}$ (broadcasted to the rank-3 tensor). That is, we have $a^{[\ell+1]} \in \mathbb{R}^{n_h^{[\ell+1]} \times n_w^{[\ell+1]} \times n_c^{[\ell+1]}}$ given by

$$a^{[\ell+1]}_{\eta}{}^k{}_l = g^{[\ell+1]}(z_\eta^{[\ell+1]})^k{}_l,$$

where

$$z_\eta^{[\ell+1]k}{}_l = F_\eta *_{s^{[\ell+1]}}^{p^{[\ell+1]}} a^{[\ell]} + b^{[\ell+1]}.$$

We remark here that the number of parameters we need to train is given by the filters with number of parameters

$$f^{[\ell+1]} \times f^{[\ell+1]} \times n_c^{[\ell]} \times n_c^{[\ell+1]},$$

plus the bias terms

$$1 \times n_c^{[\ell+1]},$$

that is,

$$\begin{aligned} \#(\text{Parameters}) &= f^{[\ell+1]} \times f^{[\ell+1]} \times n_c^{[\ell]} \times n_c^{[\ell+1]} + 1 \times n_c^{[\ell+1]} \\ &= n_c^{[\ell+1]} (n_c^{[\ell]} (f^{[\ell+1]})^2 + 1) \end{aligned}$$

12.2 Pooling Layers (**pool**)

To reduce computational cost and to help prevent over-fitting, a new type of layer is needed to reduce the overall dimensions of the input-size. This is done with a “pooling” layer. There are two main types of pooling layers that we’ll discuss here, the *max pooling* layer and the *average pooling* layer.

12.2.1 Max Pooling

Suppose

$$x = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix},$$

and we wish to apply `maxPool` with a “filter size” of $f = 2$, a stride $s = 2$ and padding $p = 0$. Then we apply the max operator to the (2×2) -submatrices moving with a stride of 2, i.e., $\text{maxPool}(x) \in \mathbb{R}^{2 \times 2}$ given by

$$\begin{aligned} \text{maxPool}(x) &= \begin{bmatrix} \max\{1, 3, 2, 9\} & \max\{2, 1, 1, 1\} \\ \max\{1, 3, 5, 6\} & \max\{2, 3, 1, 2\} \end{bmatrix} \\ &= \begin{bmatrix} 9 & 2 \\ 6 & 3 \end{bmatrix}. \end{aligned}$$

Since each layer of max pooling has 3 hyper-parameters (and no trainable parameters), we denote these via

$$\text{maxPool}_{\{f,p,s\}}(x).$$

12.2.2 Average Pooling

Suppose

$$x = \begin{bmatrix} 1 & 3 & 2 & 1 \\ 2 & 9 & 1 & 1 \\ 1 & 3 & 2 & 3 \\ 5 & 6 & 1 & 2 \end{bmatrix},$$

and we wish to apply `avPool` with a “filter size” of $f = 2$, a stride of $s = 2$ and padding $p = 0$. Then we apply the averaging operator to the (2×2) -submatrices moving with a stride of 2, i.e., $\text{avPool}(x) \in \mathbb{R}^{2 \times 2}$ given by

$$\begin{aligned}\text{avPool}(x) &= \begin{bmatrix} \mathbb{E}[\{1, 3, 2, 9\}] & \mathbb{E}[\{2, 1, 1, 1\}] \\ \mathbb{E}[\{1, 3, 5, 6\}] & \mathbb{E}[\{2, 3, 1, 2\}] \end{bmatrix} \\ &= \begin{bmatrix} 3.75 & 1.25 \\ 3.75 & 2 \end{bmatrix}.\end{aligned}$$

Since each layer of average pooling has 3 hyper-parameters (and again, no trainable parameters), we denote these via

$$\text{avPool}_{\{f,p,s\}}(x).$$

12.3 A Convolutional Network

Suppose we have a collection of images (our training set), where each image is of the form $x \in \mathbb{R}^{n_h^{[0]} \times n_w^{[0]} \times n_c^{[0]}}$. We shall denote the forward propagation from layer-0 to layer-1 via convolution as the mapping `conv(1)` which encompasses the following information:

$$\text{conv}(1) = \begin{cases} \text{filter} \\ \text{padding} \\ \text{stride} \\ \text{number of filter.} \end{cases}.$$

We similarly use `pool(1)` to encompass the following information:

$$\text{pool}(1) = \begin{cases} \text{pool type} \\ \text{filter} \\ \text{padding} \\ \text{stride.} \end{cases}$$

This yields a network architecture of the following form:

$$\begin{aligned}[x] &\xrightarrow{\text{conv}^{[1]}} [z^{[1]}] \xrightarrow{\text{pool}^{[1]}} [\zeta^{[1]}] \xrightarrow{g^{[1]}} [a^{[1]}] \xrightarrow{\text{conv}^{[2]}} [z^{[2]}] \xrightarrow{\text{pool}^{[2]}} [\zeta^{[2]}] \xrightarrow{g^{[2]}} \\ &\xrightarrow{g^{[2]}} [a^{[2]}] \xrightarrow{\text{flatten}} [a^{[2]}] \xrightarrow{\varphi^{[1]}} [z^{[3]}] \xrightarrow{g^{[3]}} [a^{[3]}] \longrightarrow \cdots \longrightarrow \hat{y}\end{aligned}$$

We remark here that the convolution and pooling layers are done before the fully connected layers. Moreover, we apply the nonlinearity after the pooling, but this doesn't matter when doing max pooling, since our nonlinearities are typically non-decreasing. We choose this order because it's typically computationally cheaper.

We also remark that since each output of a convolutional layer only depends on a subset of features, our model is less prone to over-fitting.

12.4 Backpropagation

We introduce the following tensoral notation: We say $x \in \mathbb{R}^a_{b,c}$ is a $(1,2)$ -tensor written in index form

$$x = (x^\rho_{ij})$$

with $1 \leq \rho \leq a$, $1 \leq i \leq b$ and $1 \leq j \leq c$. Similarly, we say $W \in \mathbb{R}^{a,b,c}_d$ is a $(3,1)$ -tensor written in index form

$$W = (W^{\eta ij}_\rho).$$

Suppose $x \in \mathbb{R}^{n_c}_{n_h, n_w}$, $W \in \mathbb{R}^{m_c, f, f}_{n_c}$ and $b \in \mathbb{R}^{m_c}$ with padding $p \geq 0$ and stride $s \in \mathbb{N}$. Then we have that

$$z = \text{conv}(x) \in \mathbb{R}^{m_c}_{m_h, m_w}$$

is given by

$$z^\eta_{k,l} = \sum_{\rho=1}^{n_c} \sum_{i,j=1}^f W^{\eta, i, j}_\rho x^\rho_{i+s(k-1)-p, j+s(l-1)-p} \chi_{\mathcal{I}_{k,l}}(i, j) + b^\eta.$$

This is the general formula for the forward propagation of a **conv** layer.

We now compute derivatives for general loss function \mathbb{L} :

$$\frac{\partial z^\eta_{k,l}}{\partial b^\mu} = \delta^\eta_\mu,$$

and hence

$$\begin{aligned} \frac{\partial \mathbb{L}}{\partial b^\mu} &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^\eta_{k,l}} \frac{\partial z^\eta_{k,l}}{\partial b^\mu} \\ &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^\eta_{k,l}} \delta^\eta_\mu \\ &= \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z^\mu_{k,l}}. \end{aligned}$$

Next we consider

$$\begin{aligned}\frac{\partial z_{k,l}^\eta}{\partial W_{\alpha,\mu,\nu}^\beta} &= \sum_{\rho=1}^{n_c} \sum_{i,j=1}^f \delta_\alpha^\eta \delta_\mu^i \delta_\nu^j \delta_\rho^\beta x_{i+s(k-1)-p, j+s(l-1)-p}^\rho \chi_{\mathcal{I}_{k,l}}(i, j) \\ &= \delta_\alpha^\eta x_{\mu+s(k-1)-p, \nu+s(l-1)-p}^\beta \chi_{\mathcal{I}_{k,l}}(\mu, \nu)\end{aligned}$$

and hence

$$\begin{aligned}\frac{\partial \mathbb{L}}{\partial W_{\alpha,\mu,\nu}^\beta} &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z_{k,l}^\eta} \frac{\partial z_{k,l}^\eta}{\partial W_{\alpha,\mu,\nu}^\beta} \\ &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z_{k,l}^\eta} \delta_\alpha^\eta x_{\mu+s(k-1)-p, \nu+s(l-1)-p}^\beta \chi_{\mathcal{I}_{k,l}}(\mu, \nu) \\ &= \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z_{k,l}^\alpha} x_{\mu+s(k-1)-p, \nu+s(l-1)-p}^\beta \chi_{\mathcal{I}_{k,l}}(\mu, \nu).\end{aligned}$$

Finally, we consider

$$\begin{aligned}\frac{\partial z_{k,l}^\eta}{\partial x_{\mu,\nu}^\alpha} &= \sum_{\rho=1}^{n_c} \sum_{i,j=1}^f W^{\eta,i,j}_\rho \delta_\alpha^\rho \delta_{i+s(k-1)-p}^\mu \delta_{j+s(l-1)-p}^\nu \chi_{\mathcal{I}_{k,l}}(i, j) \\ &= W^{\eta,\mu-p-s(k-1), \nu-p-s(l-1)}_\alpha \chi_{\mathcal{I}_{k,l}}(\mu-p-s(k-1), \nu-p-s(l-1)) \\ &= W^{\eta,\mu-p-s(k-1), \nu-p-s(l-1)}_\alpha \begin{cases} 1 & \text{if } p < (\mu, \nu) \leq p + (n_h, n_w) \\ 0 & \text{else} \end{cases},\end{aligned}$$

and hence

$$\begin{aligned}\frac{\partial \mathbb{L}}{\partial x_{\mu,\nu}^\alpha} &= \sum_{\eta=1}^{m_c} \sum_{k=1}^{m_h} \sum_{l=1}^{m_w} \frac{\partial \mathbb{L}}{\partial z_{k,l}^\eta} \frac{\partial z_{k,l}^\eta}{\partial x_{\mu,\nu}^\alpha} \\ &= \end{aligned}$$

Appendices

A `utils.py`

```
1 #! python3
2 import copy
3
4 import numpy as np
5 from sklearn.utils import shuffle
6
7 import activators
8 from activators import ACTIVATORS
9
10 ## Usefule printing function
11 def print_array_dict(D):
12     """
13     Parameters
14     -----
15     D : Dict[array_like]
16
17     Returns
18     -----
19     None
20     """
21     txt = "Array_{0}_has_shape_{1}\n{2}"
22     for k, v in D.items():
23         print(txt.format(str(k), v.shape, v))
24
25
26 ## Partition data into training, development, and test sets
27 def partition_data(x, y, train_ratio):
28     """
29     Parameters
30     -----
31     x : array_like
32         x.shape = (m, N)
33     y : array_like
34         y.shape = (k, N)
35     train_ratio : float
36         0<=train_ratio<=1
37
38     Returns
39     -----
40     train : Tuple[array_like]
41     dev : Tuple[array_like]
```

```

42     test : Tuple[array_like]
43     """
44     ## Shuffle the data
45     x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
46     x = x.T
47     y = y.T
48
49     ## Get the size of partitions
50     N = x.shape[1]
51     N_train = int(train_ratio * N)
52     N_mid = (N - N_train) // 2
53
54     ## Create partitions
55     train = (x[:, :N_train], y[:, :N_train])
56     dev = (x[:, N_train:N_train + N_mid], y[:, N_train:N_train + N_mid])
57     test = (x[:, N_train + N_mid:], y[:, N_train + N_mid:])
58
59     assert(x.all() == np.concatenate([train[0], dev[0], test[0]], axis=1).all())
60     assert(y.all() == np.concatenate([train[1], dev[1], test[1]], axis=1).all())
61
62     return train, dev, test
63
64 ## Partition training data into batches
65 def get_batches(x, y, b):
66     """
67     Parameters
68     -----
69     x : array_like
70         x.shape = (m, n)
71     y : array_like
72         y.shape = (k, n)
73     b : int
74
75     Returns
76     -----
77     batches : List[Dict]
78         batches[i]['x'] : array_like
79             x.shape = (m, b) # except last batch
80             y.shape = (k, b) # except last batch
81
82     """
83     m, n = x.shape
84     B = int(np.ceil(n / b))
85     batches = []
86     for i in range(B):
87         x_temp = x[:, (b * i):(b * (i + 1))]
88         y_temp = y[:, (b * i):(b * (i + 1))]

```

```

89         batches.append({'x' : x_temp, 'y' : y_temp})
90     # Slicing automatically ends at the end of
91     # the list if the stop is outside the index
92     return batches
93
94 ##### General Neural Network Model #####
95
96 ## Retrieve number of examples and layer dimensions
97 def dim_retrieval(x, y, hidden_sizes):
98     """
99     Parameters
100     -----
101     x : array_like
102         x.shape = (layers[0], n)
103     y : array_like
104         y.shape = (layers[L], n)
105     hidden_sizes : List[int]
106         hidden_sizes[i-1] = The number nodes layer i
107     Returns
108     -----
109     n : int
110         The number of training examples
111     layers : List
112         layer[l] = # nodes in layer l
113
114     """
115     m, n = x.shape
116     assert(y.shape[1] == n)
117     K = y.shape[0]
118     layers = [m]
119     layers.extend(hidden_sizes)
120     layers.append(K)
121
122     return n, layers
123
124 ## Initialize parameters using the size of each layer
125 def initialize_parameters_random(layers):
126     """
127     Parameters
128     -----
129     layers : List[int]
130         layers[l] = # nodes in layer l
131     Returns
132     -----
133     params : Dict[Dict]
134         w[l] : array_like
135             dwl.shape = (layers[l], layers[l-1])

```

```

136         b[l] : array_like
137         dbl.shape = (layers[l], 1)
138     """
139     w = {}
140     b = {}
141     for l in range(1, len(layers)):
142         w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
143         b[l] = np.zeros((layers[l], 1))
144     params = {'w' : w, 'b' : b}
145     return params
146
147 ## Forward and Backward Linear Activations
148 def linear_activation_forward(a_prev, w, b, activator):
149     """
150     Parameters
151     -----
152     a_prev : array_like
153         a_prev.shape = (layers[l], n)
154     w : array_like
155         w.shape = (layers[l+1], layers[l])
156     b : array_like
157         b.shape = (layers[l+1], 1)
158     activator : str
159         activator in ACTIVATORS
160
161     Returns
162     -----
163     z : array_like
164         z.shape = (layer_dims[l+1], n)
165     a : array_like
166         a.shape = (layer_dims[l+1], n)
167     """
168     assert activator in ACTIVATORS, f'{activator}_is_not_a_valid_activator.'
169
170     z = w @ a_prev + b
171     if activator == 'relu':
172         a, _ = activators.relu(z)
173     elif activator == 'sigmoid':
174         a, _ = activators.sigmoid(z)
175     elif activator == 'tanh':
176         a, _ = activators.tanh(z)
177     return z, a
178
179 def linear_activation_backward(delta_next, z, w, activator):
180     """
181     Parameters
182     -----

```

```

183     delta_next : array_like
184         delta_next.shape = (layers[l+1], n)
185     z : array_like
186         z.shape = (layers[l+1], n)
187     w : array_like
188         w.shape = (layers[l+1], layers[l])
189     activator : str
190         activator in ACTIVATORS
191
192     Returns
193     -----
194     delta : array_like
195         delta.shape = (layers[l], n)
196     """
197     assert activator in ACTIVATORS, f'{activator}_is_not_a_valid_activator.'
198
199     n = delta_next.shape[1]
200
201     if activator == 'relu':
202         _, dg = activators.relu(z)
203     elif activator == 'sigmoid':
204         _, dg = activators.sigmoid(z)
205     elif activator == 'tanh':
206         _, dg = activators.tanh(z)
207
208     da = w.T @ delta_next
209     assert(da.shape == (w.shape[1], n))
210     delta = da * dg
211     assert(delta.shape == (w.shape[1], n))
212     return delta
213
214
215 ## Forward and Backward Propagation with Dropout Regularization
216 # Generate dropout matrices
217 def dropout_matrices(layers, num_examples, keep_prob):
218     """
219     Parameters
220     -----
221     layers : List[int]
222         layers[l] = number of nodes in layer l
223     num_examples : int
224         The number of training examples
225     keep_prob : List[float]
226         keep_prob[l] = The probabiltly of keeping a node in layer l
227
228     Returns
229     -----

```

```

230     D : Dict[array_like]
231         D[l].shape = (layers[l], num_ex)
232         D[l] = a Boolean array
233     """
234     np.random.seed(1)
235     L = len(layers)
236     D = {}
237     for l in range(L - 1):
238         D[l] = np.random.rand(layers[l], num_examples)
239         D[l] = (D[l] < keep_prob[l]).astype(int)
240         assert(D[l].shape == (layers[l], num_examples))
241     return D
242
243 def forward_propagation_dropout(x, params, activators, D, keep_prob):
244     """
245     Parameters
246     -----
247     x : array_like
248         x.shape = (layers[0] n)
249     params : Dict[Dict]
250         params['w'][l] : array_like
251             wl.shape = (layers[l], layers[l-1])
252         params['b'][l] : array_like
253             bl.shape = (layers[l], 1)
254     activators : List[str]
255         activators[l] = activation function of layer l+1
256     D : Dict[array_like]
257         D[l].shape = (layer_dims[l], num_ex)
258         D[l] = a Boolean array astype(int)
259     keep_prob : List[float]
260         keep_prob[l] = The probabilty of keeping a node in layer l
261
262     Returns
263     -----
264     cache : Dict[Dict]
265         cache['z'][l] : array_like
266             z[l].shape = (layers[l], n)
267         cache['a'][l] : array_like
268             a[l].shape = (layers[l], n)
269     """
270     # Retrieve parameters
271     w = params['w']
272     b = params['b']
273     L = len(w) # Number of layers excluding output layer
274     n = x.shape[1]
275     # Set empty caches
276     a = {}

```



```

277     z = {}
278     # Dropout on layer 0
279     a[0] = x
280     a[0] = a[0] * D[0]
281     a[0] /= keep_prob[0]
282     # Loop through hidden layers
283     for l in range(1, L + 1):
284         z[l], al = linear_activation_forward(a[l - 1], w[l], b[l], activators[l - 1])
285         al = al * D[l]
286         al /= keep_prob[l]
287         z[l] = z[l]
288         a[l] = al
289     # Output layer
290     z[L], a[L] = linear_activation_forward(a[L - 1], w[L], b[L], activators[-1])
291
292     cache = {'z' : z, 'a' : a}
293     return cache
294
295 def backward_propagation_dropout(x, y, params, cache, activators, D, keep_prob):
296     """
297     Parameters
298     -----
299     x : array_like
300         x.shape = (layers[0], n)
301     y : array_like
302         y.shape = (layers[-1], n)
303     params : Dict[Dict[array_like]]
304         params['w'][l] : array_like
305             w[l].shape = (layers[l], layers[l-1])
306         params['b'][l] : array_like
307             b[l].shape = (layers[l], 1)
308     cache : Dict[Dict[array_like]]
309         cache['a'][l] : array_like
310             a[l].shape = (layers[l], n)
311         cache['z'][l] : array_like
312             z[l].shape = (layers[l], n)
313     activators : List[str]
314         activators[l] = activation function of layer l+1
315     D : Dict[array_like]
316         D[l].shape = (layer_dims[l], num_ex)
317         D[l] = a Boolean array astype(int)
318     keep_prob : List[float]
319         keep_prob[l] = The probabilty of keeping a node in layer l
320
321     Returns
322     -----
323     grads : Dict[Dict]

```

```

324         grads['dw'][l] : array_like
325         dw[l].shape = w[l].shape
326         grads['db'][l] : array_like
327         db[l].shape = b[l].shape
328     """
329     ## Retrieve parameters
330     a = cache['a']
331     z = cache['z']
332     w = params['w']
333     n = x.shape[1]
334     L = len(z)
335
336     ## Compute deltas
337     delta = {}
338     delta[L] = a[L] - y
339     for l in reversed(range(1, L)):
340         deltal = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activators)
341         deltal = deltal * D[l]
342         deltal /= keep_prob[l]
343         delta[l] = deltal
344
345     ## Compute gradients
346     dw = {}
347     db = {}
348
349     for l in range(1, L + 1):
350         db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
351         assert(db[l].shape == (w[l].shape[0], 1))
352         dw[l] = (1 / n) * delta[l] @ a[l - 1].T
353         assert(dw[l].shape == w[l].shape)
354     grads = {'w' : dw, 'b' : db}
355     return grads
356
357
358 ## Forward and Backward Propagation with L2-Regularization
359 def forward_propagation(x, params, activators):
360     """
361     Parameters
362     -----
363     x : array_like
364         x.shape = (layers[0] n)
365     params : Dict[Dict]
366         params['w'][l] : array_like
367             wl.shape = (layers[l], layers[l-1])
368         params['b'][l] : array_like
369             bl.shape = (layers[l], 1)
370     activators : List[str]

```

```

371         activators[l] = activation function of layer l+1
372 Returns
373 -----
374 cache : Dict[Dict]
375     cache['z'][l] : array_like
376     z[l].shape = (layers[l], n)
377     cache['a'][l] : array_like
378     a[l].shape = (layers[l], n)
379 """
380 # Retrieve parameters
381 w = params['w']
382 b = params['b']
383 L = len(w) # Number of layers excluding output layer
384 n = x.shape[1]
385 # Set empty caches
386 a = {}
387 z = {}
388 # Initialize a
389 a[0] = x
390 for l in range(1, L + 1):
391     z[l], a[l] = linear_activation_forward(a[l - 1], w[l], b[l], activators[l -
392
393     cache = {'a' : a, 'z' : z}
394     return cache
395
396 def backward_propagation(x, y, params, cache, activators, lambda_=0.0):
397     """
398     Parameters
399     -----
400     x : array_like
401         x.shape = (layers[0], n)
402     y : array_like
403         y.shape = (layers[-1], n)
404     params : Dict[Dict[array_like]]
405         params['w'][l] : array_like
406             w[l].shape = (layers[l], layers[l-1])
407         params['b'][l] : array_like
408             b[l].shape = (layers[l], 1)
409     cache : Dict[Dict[array_like]]
410         cache['a'][l] : array_like
411             a[l].shape = (layers[l], n)
412         cache['z'][l] : array_like
413             z[l].shape = (layers[l], n)
414     activators : List[str]
415         activators[l] = activation function of layer l+1
416     lambda_ : float
417         Default: 0.0

```

```

418
419 Returns
420 -----
421 grads : Dict[Dict]
422     grads['w'][l] : array_like
423     dw[l].shape = w[l].shape
424     grads['b'][l] : array_like
425     db[l].shape = b[l].shape
426 """
427 ## Retrieve parameters
428 a = cache['a']
429 z = cache['z']
430 w = params['w']
431 n = x.shape[1]
432 L = len(z)
433
434 ## Compute deltas
435 delta = {}
436 delta[L] = a[L] - y
437 for l in reversed(range(1, L)):
438     delta[l] = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activate
439
440 ## Compute gradients
441 dw = {}
442 db = {}
443 for l in range(1, L + 1):
444     db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
445     assert(db[l].shape == (w[l].shape[0], 1))
446     dw[l] = (1 / n) * (delta[l] @ a[l - 1].T + lambda_ * w[l])
447     assert(dw[l].shape == w[l].shape)
448     grads = {'w' : dw, 'b' : db}
449 return grads
450
451
452 ## Compute the (L2-regulated) cost
453 def compute_cost(y, params, cache, lambda_=0.0):
454     """
455     Parameters
456     -----
457     y : array_like
458         y.shape = (layers[-1], n)
459     params : Dict[Dict[array_like]]
460         params['w'][l] : array_like
461             w[l].shape = (layers[l], layers[l-1])
462         params['b'][l] : array_like
463             b[l].shape = (layers[l], 1)
464     cache : Dict[Dict[array_like]]

```

```

465         cache['z'][l] : array_like
466         z[l].shape = (layers[l], n)
467         cache['a'][l] : array_like
468         a[l].shape = (layers[l], n)
469     lambda_ : float
470         Default: 0.0
471
472     Returns
473     -----
474     cost : float
475         The cost evaluated at y and aL
476     """
477     ## Retrieve parameters
478     n = y.shape[1]
479     a = cache['a']
480     w = params['w']
481     L = len(a)
482     aL = a[L - 1]
483
484     ## Regularization term
485     R = 0
486     for l in range(1, L):
487         R += np.sum(w[l] * w[l])
488     R *= (lambda_ / (2 * n))
489
490     ## Unregularized cost
491     J = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
492
493     ## Total Cost
494     cost = J + R
495     cost = float(np.squeeze(cost))
496     return cost
497
498
499 ## Update parameters via gradient descent
500 def update_parameters(params, grads, learning_rate=0.01):
501     """
502     Parameters
503     -----
504     params : Dict[Dict]
505         params['w'][l] : array_like
506             w[l].shape = (layers[l], layers[l-1])
507         params['b'][l] : array_like
508             b[l].shape = (layers[l], 1)
509     grads : Dict[Dict]
510         grads['w'][l] : array_like
511             dw[l].shape = w[l].shape

```

```

512         grads['b'][l] : array_like
513         db[l].shape = b[l].shape
514     learning_rate : float
515         Default: 0.01
516         The learning rate for gradient descent
517
518     Returns
519     -----
520     params : Dict[Dict]
521         params['w'][l] : array_like
522             w[l].shape = (layers[l], layers[l-1])
523         params['b'][l] : array_like
524             b[l].shape = (layers[l], 1)
525     """
526     ## Retrieve parameters
527     w = copy.deepcopy(params['w'])
528     b = copy.deepcopy(params['b'])
529     L = len(w)
530
531     ## Retrieve gradients
532     dw = grads['w']
533     db = grads['b']
534
535     ## Perform update
536     for l in range(1, L + 1):
537         w[l] = w[l] - learning_rate * dw[l]
538         b[l] = b[l] - learning_rate * db[l]
539
540     params = {'w' : w, 'b' : b}
541     return params
542
543
544
545
546
547
548
549
550
551 ##### Dropout NN Model #####
552 def model_nn(x, y, hidden_layer_sizes, activators, keep_prob=1.0, num_iters=10000, p
553     """
554     Parameters
555     -----
556     x : array_like
557         x.shape = (layers[0], n)
558     y : array_like

```

```

559         y.shape = (layers[-1], n)
560     hidden_layer_sizes : List[int]
561         The number nodes layer l = hidden_layer_sizes[l-1]
562     activators : List[str]
563         activators[l] = activation function of layer l+1
564     keep_prob : List[float] | float
565         keep_prob[l] = The probability of keeping a node in layer l
566         keep_prob = The same probability for all input and hidden layers
567     num_iters : int
568         Number of iterations with which our model performs gradient descent
569     print_cost : Boolean
570         If True, print the cost every 1000 iterations
571
572     Returns
573     -----
574     params : Dict[Dict]
575         params['w'][l] : array_like
576             w[l].shape = (layers[l], layers[l-1])
577         params['b'][l] : array_like
578             b[l].shape = (layers[l], 1)
579     cost : float
580         The final cost value for the optimized parameters returned
581     """
582     ## Set dimensions and Initialize parameters
583     n, layers = dim_retrieval(x, y, hidden_layer_sizes)
584     params = initialize_parameters_random(layers)
585
586     ## Expand keep_prob to a list if it's a single float
587     if isinstance(keep_prob, float):
588         keep_prob = [keep_prob] * (len(layers) - 1)
589
590     # main gradient descent loop
591     for i in range(num_iters):
592         D = dropout_matrices(layers, n, keep_prob)
593         cache = forward_propagation(x, params, activators, D, keep_prob)
594         cost = compute_cost(cache, y)
595         grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
596         params = update_parameters(params, grads)
597
598         if print_cost and i % 1000 == 0:
599             print(f'Cost_after_iteration_{i}:_{cost}')
600
601     return params, cost
602
603
604
605

```

```

606
607
608
609 ##### TESTING #####
610 def test_dropout_nn():
611     x = np.random.rand(4, 500)
612     y = np.random.rand(1, 500)
613     hidden_layer_sizes = [4, 5, 4]
614     activators = ['relu', 'relu', 'relu', 'sigmoid']
615     keep_prob = 1.0
616     params, cost = model_nn(x, y, hidden_layer_sizes, activators, keep_prob)
617     print(params)
618
619
620
621 ##### Functions to use later
622 def reshape_labels(num_labels, y):
623     """
624     Parameters
625     -----
626     num_labels : int
627         The number of possible labels the output y may take
628     y : array_like
629         y.size = n
630         y[i] takes values in {1,2,...,num_labels}
631     Returns
632     Y : array_like
633         Y.shape = (num_labels, n)
634         Y[i][j] = 1 if y[j] = i, Y[i][j] = 0 otherwise
635     -----
636     """
637
638     if num_labels <= 2:
639         return y
640     else:
641         omega = []
642         for i in range(num_labels):
643             omega.append(np.eye(1, num_labels, i)) # the standard i-th basis vector
644
645         Y = np.concatenate([omega[i] for i in y], axis=0).T
646         for i in range(num_labels):
647             for j in range(n):
648                 if y[j] == i:
649                     assert Y[i][j] == 1
650                 else:
651                     assert Y[i][j] == 0
652         return Y

```



```

653
654 #####
655 if __name__ == '__main__':
656     test_dropout_nn()

```

B activators.py

```

1 import numpy as np
2
3 ACTIVATORS = ['relu', 'sigmoid', 'tanh']
4
5 ## Activator functions
6 # The (leaky-)ReLU function
7 def relu(z, beta=0.0):
8     """
9     Parameters
10    -----
11    z : array_like
12    beta : float
13
14    Returns
15    -----
16    r : array_like
17        The (broadcasted) ReLU function when beta=0, the leaky-ReLU otherwise.
18    dr : array_like
19        The (broadcasted) derivative of the (leaky-)ReLU function
20    """
21    # Change scalar to array if needed
22    z = np.array(z)
23    # Compute value of ReLU(z)
24    r = np.maximum(z, beta * z)
25    # Compute differential ReLU'(z)
26    dr = ((~(z < 0)) * 1) + ((z < 0) * beta)
27    return r, dr
28
29 # The sigmoid function
30 def sigmoid(z):
31     """
32     Parameters
33    -----
34    z : array_like
35
36    Returns
37    -----
38    sigma : array_like
39        The (broadcasted) value of the sigmoid function evaluated at z

```

```

40     dsigma : array_like
41     """ The (broadcasted) derivative of the sigmoid function evaluate at z
42     """
43     # Compute value of sigmoid
44     sigma = (1 / (1 + np.exp(-z)))
45     # Compute differential of sigmoid
46     dsigma = sigma * (1 - sigma)
47     return sigma, dsigma
48
49 # The hyperbolic tangent function
50 def tanh(z):
51     """
52     Parameters
53     -----
54     z : array_like
55
56     Returns
57     phi : array_like
58         The (broadcasted) value of the hyperbolic tangent function evaluated at z
59     dphi : array_like
60         The (broadcasted) derivative of hyperbolic tangent function evaluated at z
61     """
62     # Compute value of tanh
63     phi = np.tanh(z)
64     # Compute differential of tanh
65     dphi = 1 - (phi * phi)
66     return phi, dphi

```

C The Normalization Operator

In this section, we wish to character the (reverse) differential of the normalization operator $N : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$ given in coordinates by

$$N : x_j^i \mapsto \frac{x_j^i - \mathbb{E}[x^i]}{\sqrt{\mathbb{V}[x^i] + \epsilon}}.$$

First, let's rewrite this without coordinates

$$\begin{aligned} N(x) &= (\mathbb{V}[x] + \epsilon)^{\odot -\frac{1}{2}} \vec{1}^T \odot (x - \mathbb{E}[x] \vec{1}^T) \\ &=: f(x) \odot g(x). \end{aligned}$$

Now, let's fix $(x, \xi) \in T\mathbb{R}^{m \times n}$, and we immediately see that the Hadamard product obeys the Leibniz Rule with exterior differentiation, i.e.,

$$dN_x(\xi) = g(x) \odot df_x(\xi) + f(x) \odot dg_x(\xi),$$

so we consider these computations separately. Moreover, we now need to compute the derivative of the expectation \mathbb{E} and variance \mathbb{V} operators.

1. For the expectation of a random vector, $\mathbb{E} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^m$, we first rewrite \mathbb{E} as follows

$$\begin{aligned} \mathbb{E}[x] &= \sum_{i=1}^m \left(\frac{1}{n} \sum_{j=1}^n x_j^i \right) e_i \\ &= \frac{1}{n} x \cdot \vec{1} \\ &= \frac{1}{n} R_{\vec{1}}(x) \end{aligned}$$

where $\vec{1} = (1, 1, \dots, 1) \in \mathbb{R}^n$. This is clearly linear, so for $(x, \xi) \in T\mathbb{R}^{m \times n}$, we have that

$$\begin{aligned} d\mathbb{E}_x(\xi) &= \mathbb{E}[\xi] \\ &= \frac{1}{n} R_{\vec{1}}(\xi). \end{aligned}$$

For a fixed $x \in \mathbb{R}^{m \times n}$, we let $\mu := \mathbb{E}[x] \in \mathbb{R}^m$ denote the output.

2. For the variance of a random vector, $\mathbb{V} : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^m$, we rewrite \mathbb{V} as follows

$$\begin{aligned}\mathbb{V}[x] &= \sum_{i=1}^m \left(\frac{1}{n} \sum_{j=1}^n (x_j^i - \mu^i)^2 \right) e_i \\ &= \mathbb{E} \left[(x - \mu \vec{1}^T) \odot (x - \mu \vec{1}^T) \right] \\ &= \mathbb{E} \left[(x - \mu \vec{1}^T)^{\odot 2} \right].\end{aligned}$$

From the first calculation, we know how to compute the derivative of \mathbb{E} , so we focus on the input $(x - \mu \vec{1}^T)^{\odot 2}$.

To this end, we define $\psi : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$ to be the inner-most function given by

$$\begin{aligned}\psi(x) &= x - \mathbb{E}[x] \vec{1}^T \\ &= (\mathbb{1}_{\mathbb{R}^{m \times n}} - R_{\vec{1}^T} \circ \mathbb{E})(x),\end{aligned}$$

which is clearly linear. Then for $(x, \xi) \in T\mathbb{R}^{m \times n}$ we see that

$$d\psi_x(\xi) = \left(\mathbb{1}_{T_x \mathbb{R}^{m \times n}} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \right) (\xi),$$

where we used our previous computation for $d\mathbb{E}_x$.

Next, define $\phi : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}$ to be the Hadamard-square, i.e.,

$$\phi(x) = x^{\odot 2} = x \odot x.$$

Using our previous remark of the Leibniz Rule in regard to the Hadamard product, we see that for $(x, \xi) \in T\mathbb{R}^{m \times n}$,

$$\begin{aligned}d\phi_x(\xi) &= x \odot \xi + \xi \odot x \\ &= 2x \odot \xi \\ &= \odot_{2x}(\xi).\end{aligned}$$

Finally, by the compositional definition of \mathbb{V} ,

$$\mathbb{V}[x] = \mathbb{E} \circ \phi \circ \psi(x),$$

we compute for any $(x, \xi) \in T\mathbb{R}^{m \times n}$ that

$$\begin{aligned}
d\mathbb{V}_x(\xi) &= d\mathbb{E}_{\phi(\psi(x))} \circ d\phi_{\psi(x)} \circ d\psi_x(\xi) \\
&= d\mathbb{E}_{\phi(\psi(x))} \circ d\phi_{\psi(x)} \left(\xi - \frac{1}{n} \xi \vec{1} \vec{1}^T \right) \\
&= d\mathbb{E}_{\phi(\psi(x))} \left(2(x - \mu \vec{1}^T) \odot \left(\xi - \frac{1}{n} \xi \vec{1} \vec{1}^T \right) \right) \\
&= \mathbb{E} \left[2(x - \mu \vec{1}^T) \odot \left(\xi - \frac{1}{n} \xi \vec{1} \vec{1}^T \right) \right] \\
&= \mathbb{E} \left[2(x - \mu \vec{1}^T) \odot \xi \right] - 2\mathbb{E} \left[(x - \mu \vec{1}^T) \odot (\mathbb{E}[\xi] \vec{1}^T) \right].
\end{aligned}$$

Next, we notice that if we let $\gamma := \mathbb{E}[\xi] \in \mathbb{R}^m$, then

$$\begin{aligned}
\gamma \vec{1}^T &= \begin{pmatrix} \gamma^1 \\ \vdots \\ \gamma^m \end{pmatrix} (1 \quad 1 \quad \cdots \quad 1) \\
&= \begin{bmatrix} \gamma^1 & \gamma^1 & \cdots & \gamma^1 \\ \gamma^2 & \gamma^2 & \cdots & \gamma^2 \\ \vdots & \vdots & \ddots & \vdots \\ \gamma^m & \gamma^m & \cdots & \gamma^m \end{bmatrix} \in \mathbb{R}^{m \times n}
\end{aligned}$$

and hence that

$$\begin{aligned}
\mathbb{E}[(x - \mu \vec{1}^T) \odot \gamma \vec{1}^T] &= \sum_{i=1}^m \left(\frac{1}{n} \sum_{j=1}^n (x_j^i - \mu^i) \gamma^i \right) e_i \\
&= \sum_{i=1}^m (\gamma^i (\mathbb{E}[x^i] - \mu^i)) e_i \\
&= 0.
\end{aligned}$$

Resuming our computation, we now have that

$$\begin{aligned}
d\mathbb{V}_x(\xi) &= \mathbb{E} \left[2(x - \mu \vec{1}^T) \odot \xi \right] \\
&= \frac{2}{n} R_{\vec{1}} \circ \odot_{x - \mu \vec{1}^T}(\xi)
\end{aligned}$$

We remark that for a fixed $x \in \mathbb{R}^{m \times n}$, we let $\sigma^2 := \mathbb{V}[x]$ denote the output.

We have now computed the following differentials for any $(x, \xi) \in T\mathbb{R}^{m \times n}$,

$$\begin{aligned} d\mathbb{E}_x(\xi) &= \frac{1}{n} R_{\vec{1}}(\xi), \\ d\mathbb{V}_x(\xi) &= \frac{2}{n} R_{\vec{1}} \circ \odot_{x - \mu \vec{1}^T}(\xi), \end{aligned}$$

and are now ready to compute the differentials of our previously defined f and g , that is,

$$f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}, \quad f(x) = (\mathbb{V}[x] + \vec{\epsilon})^{\odot -\frac{1}{2}} \vec{1}^T,$$

and

$$g : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{m \times n}, \quad g(x) = x - \mathbb{E}[x] \vec{1}^T.$$

However, we see here that $g \equiv \psi$ as defined, and so for any $(x, \xi) \in T\mathbb{R}^{m \times n}$, we have that

$$dg_x(\xi) = \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \right) (\xi) = \xi - \frac{1}{n} \xi \vec{1} \vec{1}^T.$$

Hence we need only focus on f . To this end, for $(x, \xi) \in T\mathbb{R}^{m \times n}$, we first compute the differential of the Hadamard-root operator, $h(x) = x^{\odot -\frac{1}{2}}$,

$$\begin{aligned} dh_x(\xi) &= \frac{d}{dt} \Big|_{t=0} (x + t\xi)^{\odot -\frac{1}{2}} \\ &= \frac{d}{dt} \Big|_{t=0} \left[(x_j^i + t\xi_j^i)^{-\frac{1}{2}} \right] \\ &= \left[-\frac{1}{2} (x_j^i)^{-\frac{3}{2}} \xi_j^i \right] \\ &= -\frac{1}{2} x^{\odot -\frac{3}{2}} \odot \xi \\ &= -\frac{1}{2} \odot_{x^{\odot -\frac{3}{2}}}(\xi). \end{aligned}$$

After writing f as the composition

$$f(x) = R_{\vec{1}^T} \circ h(\mathbb{V}[x] + \vec{\epsilon}),$$

we now compute

$$\begin{aligned} df_x(\xi) &= R_{\vec{1}^T} \circ dh_{\sigma^2 + \vec{\epsilon}} \circ d\mathbb{V}_x(\xi) \\ &= -\frac{1}{n} R_{\vec{1}^T} \circ \odot_{(\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}}} \circ R_{\vec{1}} \circ \odot_{x - \mu \vec{1}^T}(\xi) \\ &= -\frac{1}{n} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}} \vec{1}^T \odot (x - \mu \vec{1}^T) \odot \xi \vec{1} \end{aligned}$$

Finally, recalling that we defined

$$N(x) = f(x) \odot g(x),$$

and so we have that

$$\begin{aligned} dN_x(\xi) &= g(x) \odot df_x(\xi) + f(x) \odot dg_x(\xi) \\ &= -\frac{1}{n} \odot_{x-\mu\vec{1}^T} R_{\vec{1}^T} \circ \odot_{(\sigma^2+\vec{\epsilon})^{\odot-\frac{3}{2}}} \circ R_{\vec{1}} \circ \odot_{x-\mu\vec{1}^T}(\xi) \\ &\quad + (\sigma^2 + \vec{\epsilon})^{\odot-\frac{1}{2}} \vec{1}^T \odot \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \right) (\xi). \end{aligned}$$

To simplify the expression for implementation in python, we make the auxiliary definitions (which only depend on the forward propagating computations)

$$y := N(x),$$

and

$$\theta := (\sigma^2 + \vec{\epsilon})^{\odot-\frac{1}{2}}.$$

Then our computation reduces to

$$dN_x(\xi) = -\frac{1}{n} (x - \mu\vec{1}^T) \odot$$

$$\begin{aligned} dN_x(\xi) &= -\frac{1}{n} \eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta(\xi) + \theta \circ \left(\mathbb{1} - \frac{1}{n} R_{\vec{1}^T} \circ R_{\vec{1}} \right) (\xi) \\ &= \left[-\frac{1}{n} \eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta + \theta - \frac{1}{n} \theta \circ R_{\vec{1}^T} \circ R_{\vec{1}} \right] (\xi). \end{aligned}$$

Then for $\zeta \in T_{N(x)}\mathbb{R}^{m \times n}$, we have the reverse differential

$$\begin{aligned} \langle rN_x(\zeta), \xi \rangle_F &= \langle \zeta, dN_x(\xi) \rangle_F \\ &= \left\langle \left[-\frac{1}{n} \eta \circ R_{\vec{1}^T} \circ \Theta \circ R_{\vec{1}} \circ \eta + \theta - \frac{1}{n} \theta \circ R_{\vec{1}^T} \circ R_{\vec{1}} \right] (\zeta), \xi \right\rangle_F. \end{aligned}$$

C.1 The Normalization Operator v.2

Suppose $N : (\mathbb{R}^m)^n \rightarrow (\mathbb{R}^m)^n$ is given by

$$N(x_1, \dots, x_n) = (y_1, \dots, y_n),$$

where

$$y_j = \frac{x_j - \mathbb{E}[x]}{\sqrt{\mathbb{V}[x] + \epsilon}}.$$

Then for $(x, \xi) \in T(\mathbb{R}^m)^n$, we have that

$$dN_x(\xi) = \bigoplus_{j=1}^n d_j N_x(\xi_j).$$

For what follows, we fix $x \in (\mathbb{R}^m)^n$, $\alpha, \beta \in \{1, \dots, n\}$, and let $\xi \in T_{x_\alpha} \mathbb{R}^m$ and consider

$$d_\alpha y_x(\xi),$$

where

$$y := y_\beta : (\mathbb{R}^m)^n \rightarrow \mathbb{R}^m.$$

To this end, if we let

$$\mu := \mathbb{E}[x], \quad \sigma^2 := \mathbb{V}[x],$$

and consider y written compositionally as

$$y : (\mathbb{R}^m)^n \times \mathbb{R}^m \times \mathbb{R}^m \rightarrow \mathbb{R}^m, \quad y(x, \mu, \sigma^2) = (\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot (x_\beta - \mu),$$

then by the chain rule it follows that

$$d_\alpha y_x(\xi) = d_\alpha y_{(x, \mu, \sigma^2)}(\xi) + d_\mu y_{(x, \mu, \sigma^2)} \circ d_\alpha \mathbb{E}_x(\xi) + d_{\sigma^2} y_{(x, \mu, \sigma^2)} \circ d_\alpha \mathbb{V}_x(\xi).$$

Computing these differentials yields

$$\begin{aligned} d_\alpha y_{(x, \mu, \sigma^2)}(\xi) &= \delta_{\alpha\beta} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot \xi \\ d_\mu y_{(x, \mu, \sigma^2)}(\xi) &= -(\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} \odot \xi \\ d_{\sigma^2} y_{(x, \mu, \sigma^2)}(\xi) &= -\frac{1}{2} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}} \odot (x_\beta - \mu) \odot \xi \\ d_\alpha \mathbb{E}_x(\xi) &= \frac{1}{n} \xi \\ d_\alpha \mathbb{V}_x(\xi) &= \frac{2}{n} (x_\alpha - \mu) \odot \xi. \end{aligned}$$

Substituting in these differentials, we see that

$$d_\alpha (y_\beta)_x(\xi) = \left[\delta_{\alpha\beta} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} - \frac{1}{n} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{1}{2}} - \frac{1}{n} (\sigma^2 + \vec{\epsilon})^{\odot -\frac{3}{2}} \odot (x_\beta - \mu) \odot (x_\alpha - \mu) \right] \odot \xi,$$

and noting that derivative only acts via the Hadamard-product, we may conclude that the reverse derivative coincides with the usual derivative, i.e.,

$$r_\alpha(y_\beta)_x \cong d_\alpha(y_\beta)_x,$$

after the usual identification of tangent spaces. To simplify this expression, we define the constant (with respect to the tangent space)

$$\theta = (\sigma^2 + \epsilon)^{\odot -\frac{1}{2}},$$

which leads us to write

$$d_\alpha(y_\beta)_x(\xi) = [\delta_{\alpha\beta}\theta - \frac{1}{n}\theta - \frac{1}{n}\theta \odot y_\alpha \odot y_\beta] \odot \xi.$$

Moreover, since

$$d(y_\beta)_x(\xi) = \sum_{\alpha=1}^n d_\alpha(y_\beta)_x(\xi_\alpha), \quad \xi_\alpha \in T_{x_\alpha}\mathbb{R}^m,$$

it follows that for $\zeta_\beta \in T_{y_\beta}\mathbb{R}^m$, that

$$\begin{aligned} \langle r(y_\beta)_x(\zeta_\beta), \xi \rangle_{(\mathbb{R}^m)^n} &= \langle \zeta_\beta, d(y_\beta)_x(\xi) \rangle_{T_{y_\beta}\mathbb{R}^m} \\ &= \left\langle \zeta_\beta, \sum_{\alpha=1}^n d_\alpha(y_\beta)_x(\xi_\alpha) \right\rangle_{T_{y_\beta}\mathbb{R}^m} \\ &= \sum_{\alpha=1}^n \langle r_\alpha(y_\beta)_x(\zeta_\beta), \xi_\alpha \rangle_{T_{x_\alpha}\mathbb{R}^m} \\ &= \left\langle \bigoplus_{\alpha=1}^n r_\alpha(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n}, \end{aligned}$$

and hence that

$$r(y_\beta)_x(\zeta_\beta) = \bigoplus_{\alpha=1}^n r_\alpha(y_\beta)_x(\zeta_\beta).$$

Next, for $(x, \xi) \in T(\mathbb{R}^m)^n$ and $\zeta \in T_y(\mathbb{R}^m)^n$, we have that

$$\begin{aligned}
\langle rN_x(\zeta), \xi \rangle_{(\mathbb{R}^m)^n} &= \langle \zeta, dN_x(\xi) \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \zeta, \bigoplus_{\beta=1}^n d(y_\beta)_x(\xi) \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^n \langle \zeta_\beta, d(y_\beta)_x(\xi) \rangle_{T_{y_\beta} \mathbb{R}^m} \\
&= \sum_{\beta=1}^n \sum_{\alpha=1}^n \langle \zeta_\beta, d_\alpha(y_\beta)_x(\xi_\alpha) \rangle_{T_{y_\beta} \mathbb{R}^m} \\
&= \sum_{\beta=1}^n \sum_{\alpha=1}^n \langle r_\alpha(y_\beta)_x(\zeta_\beta), \xi_\alpha \rangle_{T_{x_\alpha} \mathbb{R}^m} \\
&= \sum_{\beta=1}^n \left\langle \bigoplus_{\alpha=1}^n r_\alpha(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^n \langle r(y_\beta)_x(\zeta_\beta), \xi \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \sum_{\beta=1}^n r(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n}.
\end{aligned}$$

That is,

$$\begin{aligned}
rN_x(\zeta) &= \sum_{\beta=1}^n r(y_\beta)_x(\zeta_\beta) \\
&= \bigoplus_{\alpha=1}^n \left\{ \sum_{\beta=1}^n r_\alpha(y_\beta)_x(\zeta_\beta) \right\} \\
&= \bigoplus_{\alpha=1}^n \left\{ \sum_{\beta=1}^n \left[\delta_{\alpha\beta} \theta \odot \zeta_\beta - \frac{1}{n} \theta \odot \zeta_\beta - \frac{1}{n} \theta \odot y_\alpha \odot y_\beta \odot \zeta_\beta \right] \right\} \\
&= \bigoplus_{\alpha=1}^n \left\{ \theta \odot \zeta_\alpha - \frac{1}{n} \theta \odot \sum_{\beta=1}^n \zeta_\beta - \frac{1}{n} \theta \odot y_\alpha \odot \sum_{\beta=1}^n y_\beta \odot \zeta_\beta \right\} \\
&= \bigoplus_{\alpha=1}^n \underbrace{\left\{ \theta \odot (\zeta e_\alpha) - \frac{1}{n} \theta \odot (\zeta \vec{1}) - \frac{1}{n} \theta \odot y_\alpha \odot (y \odot \zeta) \vec{1} \right\}}_{=r_\alpha N_x(\zeta)}.
\end{aligned}$$

We note here that rN_x is a rank $(2, 2)$ -tensor, and as such we need to compute its components if we wish to implement this in python. To this end, let $\{E_i^j\}$ denote the basis for $\mathbb{R}^{m \times n}$, where

$$(E_i^j)_l^k = \delta_i^k \delta_l^j,$$

and let $\{\epsilon_j\}$, $\{e_j\}$ denote the standard bases for \mathbb{R}^m and \mathbb{R}^n , respectively. We now compute

$$\begin{aligned}
rN_x(E_i^j) &= \bigoplus_{l=1}^n \left\{ \theta \odot (E_i^j e_l) - \frac{1}{n} \theta \odot (E_i^j \vec{1}) - \frac{1}{n} \theta \odot y_l \odot (y \odot E_i^j) \vec{1} \right\} \\
&= \bigoplus_{l=1}^n \left\{ \theta^k \delta_i^k \delta_l^j \epsilon_k - \frac{1}{n} \theta^k \delta_i^k \epsilon_k - \frac{1}{n} \theta^k y_l^k y_j^k \delta_i^k \epsilon_k \right\} \\
&= \bigoplus_{l=1}^n \theta^k \left\{ \delta_i^k \delta_l^j - \frac{1}{n} \delta_i^k (1 - y_l^k y_j^k) \right\} \epsilon_k \\
&= \theta^k \left[\delta_i^k \delta_l^j - \frac{1}{n} \delta_i^k (1 - y_l^k y_j^k) \right] E_k^l \quad \text{definition of direct sum} \\
&= \theta^k [\delta_i^k \delta_l^j - z^k_{lj} \delta_i^k] E_k^l,
\end{aligned}$$

that is, if ζ_j^i is a matrix, we yield the matrix

$$rN_x(\zeta) = [\theta^k [\delta_i^k \delta_l^j - z^k_{lj} \delta_i^k] \zeta_j^i]_l^k,$$

which is easily implemented in python via numpy's "einsum" function.

References

- [1] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.