

Neural Networks

Matt R

July 19, 2022

Contents

I	Neural Networks and Deep Learning	3
1	Logistic Regression	4
1.1	The Gradient	5
1.2	Implementation in Python via <code>numpy</code>	8
1.3	Implementation in Python via <code>sklearn</code>	12
2	Neural Networks: A Single Hidden Layer	14
2.1	Activation Functions	16
2.1.1	The Sigmoid Function	16
2.1.2	The Hyperbolic Tangent Function	17
2.1.3	The Rectified Linear Unit Function	17
2.1.4	The Softmax Function	18
2.2	Backward Propagation	19
3	Deep Neural Networks	25
3.1	Implementation in Python via <code>numpy</code>	27
3.2	Implementation in Python via <code>tensorflow</code>	31
II	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization	34
4	Training, Development and Test Sets	35
4.1	Python Implementation	37

5	Regularization	40
5.1	(Inverted) Dropout Regularization	41
5.1.1	Python Implementation	42
5.2	Data Augmentation	47
5.3	Early Stopping	48
6	Gradients and Numerical Remarks	49
6.1	Numerical Gradient Checking	49
6.2	Python Implementation via <code>numpy</code>	50
7	Gradient Descent	53
7.0.1	Python Implementation via <code>numpy</code>	55
7.1	Weighted Averages	61
7.2	Gradient Descent with Momentum	64
7.2.1	Python Implementation via <code>numpy</code>	65
7.3	Root Mean Squared Propagation (RMSProp)	72
7.3.1	Python Implementation via <code>numpy</code>	74
7.4	Adaptive Moment Estimation: The Adam Algorithm	81
7.4.1	Python Implementation via <code>numpy</code>	82
7.5	Learning Rate Decay	89
7.6	Python Implementation via <code>numpy</code>	90
8	Tuning Hyper-Parameters	98
9	Batch Normalization	100
9.1	Backward Propagation	102
9.2	Backward Propagation - OLD	108
9.3	Inferencing	114
9.4	Algorithm Outline	115
9.5	Python Implementation	116
Appendix A The Reverse Differential		118
Appendix B The Normalization Operator		123
B.1	The Normalization Operator v.2	127
References		132

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 $x \in \mathbb{R}^{n \times N}$ 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 $w \in \mathbb{R}^{1 \times n}$, $b \in \mathbb{R}$, and let

$$a = \sigma(wx + b).$$

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 an 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

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

1.1 The Gradient

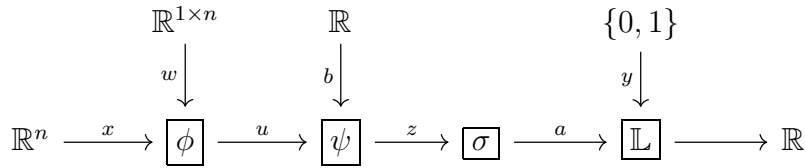
We wish to compute the gradient of our cost function \mathbb{J} with respect to our trainable parameters, $w \in \mathbb{R}^{1 \times n}$ and $b \in \mathbb{R}$. To this end, we define the functions

$$\phi : \mathbb{R}^{1 \times n} \times \mathbb{R}^n \rightarrow \mathbb{R}, \quad \phi(w, x) = wx,$$

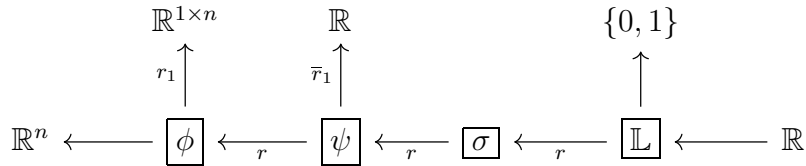
and

$$\psi : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}, \quad \psi(b, u) = u + b.$$

Then our logistic regression model for a single example follows the following network layout:



Let's now analyze our reverse differentials for this type of composition:



1.

$$\phi : \mathbb{R}^{1 \times n} \times \mathbb{R}^n \rightarrow \mathbb{R}, \quad u := \phi(w, x) = wx.$$

Then for any $(w, x) \in \mathbb{R}^{1 \times n} \times \mathbb{R}^n$ and any $\eta \in T_w \mathbb{R}^{1 \times n}$, we have that

$$\begin{aligned} d_1 \phi_{(w, x)}(\eta) &= \eta x \\ &= R_x(\eta), \end{aligned}$$

where R_x is the right-multiplication operator. It then follows that for any $\zeta \in T_u \mathbb{R}$, that

$$\begin{aligned} \langle r_1 \phi_{(w, x)}(\zeta), \eta \rangle_{\mathbb{R}^{1 \times n}} &= \langle \zeta, d_1 \phi_{(w, x)}(\eta) \rangle_{\mathbb{R}} \\ &= \langle \zeta, R_x(\eta) \rangle_{\mathbb{R}} \\ &= \langle R_{x^T}(\zeta), \eta \rangle_{\mathbb{R}^{1 \times n}}, \end{aligned}$$

and hence that

$$r_1 \phi_{(w, x)} = R_{x^T}.$$

2.

$$\psi : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}, \quad z := \psi(b, u) = u + b.$$

Then for any $(b, u) \in \mathbb{R} \times \mathbb{R}$ and any $\xi \in T_u \mathbb{R}$, we have that

$$d\psi_{(b, u)}(\xi) = \mathbf{1}_{\mathbb{R}}(\xi),$$

and similarly for any $\eta \in T_b \mathbb{R}$, we have that

$$\bar{d}_1 \psi_{(b, u)}(\eta) = \mathbf{1}_{\mathbb{R}}(\eta).$$

We then immediately have that

$$r\psi_{(b, u)} = \mathbf{1}_{\mathbb{R}},$$

and

$$\bar{r}_1 \psi_{(b, u)} = \mathbf{1}_{\mathbb{R}}.$$

3.

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

Then

$$\begin{aligned}
r\sigma_z &= \frac{e^{-z}}{(1 + e^{-z})^2} \\
&= \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \\
&= \sigma(z) \frac{1 + e^{-z} - 1}{1 + e^{-z}} \\
&= \sigma(z)(1 - \sigma(z)) \\
&= a(1 - a).
\end{aligned}$$

4.

$$\mathbb{L} : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}, \quad \mathbb{L}(a, y) = -[y \log(a) + (1 - y) \log(1 - a)].$$

Then

$$r\mathbb{L}_{(a,y)} = -\frac{y}{a} + \frac{1-y}{1-a}$$

We now compute the gradients with respect to w and b . To this end,

$$\begin{aligned}
\frac{\partial \mathbb{J}}{\partial w} &= \frac{1}{N} \sum_{j=1}^N r_1 \phi_{w, x_j} \circ r\psi_{(b, u_j)} \circ r\sigma_{z_j} \circ r\mathbb{L}_{(a_j, y_j)} \\
&= \frac{1}{N} \sum_{j=1}^N R_{x_j^T} \circ \left[-\frac{y_j}{a_j} + \frac{1-y_j}{1-a_j} \right] \cdot (a_j(1-a_j)) \\
&= \frac{1}{N} \sum_{j=1}^N (a_j - y_j) x_j^T \\
&= \frac{1}{N} (a - y) x^T,
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial \mathbb{J}}{\partial b} &= \frac{1}{N} \sum_{j=1}^N \bar{r}_1 \psi_{b, u_j} \circ r\sigma_{z_j} \circ r\mathbb{L}_{(a_j, y_j)} \\
&= \frac{1}{N} \sum_{j=1}^N (a_j - y_j)
\end{aligned}$$

1.2 Implementation in Python via numpy

Here we include the general method of coding a logistic regression model with L^2 -regularization via the classical numpy library.

```
1  #! python3
2
3  import numpy as np
4
5  from mllib.utils import apply_activation
6
7
8  class LinearParameters:
9      def __init__(self, dims, bias=True, seed=1):
10         """
11         Parameters:
12         -----
13         dims : tuple(int, int)
14         bias : Boolean
15             Default : True
16         seed : int
17             Default : 1
18
19         Returns:
20         -----
21         None
22         """
23         np.random.seed(seed)
24         self.dims = dims
25         self.bias = bias
26         self.w = np.random.randn(*dims) * 0.01
27         if bias:
28             self.b = np.zeros((dims[0], 1))
29
30     def forward(self, x):
31         """
32         Parameters:
33         -----
34         x : array_like
35
36         Returns:
37         -----
38         z : array_like
39         """
40         z = np.einsum("ij,jk", self.w, x)
41         if self.bias:
42             z += self.b
```



```

43
44         return z
45
46     def backward(self, dz, x):
47         """
48         Parameters:
49         -----
50         dz : array_like
51         x : array_like
52
53         Returns:
54         -----
55         None
56         """
57         if self.bias:
58             self.db = np.sum(dz, axis=1, keepdims=True)
59             assert self.db.shape == self.b.shape
60
61             self.dw = np.einsum("ij,kj", dz, x)
62             assert self.dw.shape == self.w.shape
63
64     def update(self, learning_rate=0.01):
65         """
66         Parameters:
67         -----
68         learning_rate : float
69             Default : 0.01
70
71         Returns:
72         -----
73         None
74         """
75         w = self.w - learning_rate * self.dw
76         self.w = w
77
78         if self.bias:
79             b = self.b - learning_rate * self.db
80             self.b = b
81
82
83     class LogisticRegression:
84         def __init__(self, lp_reg):
85             """
86             Parameters:
87             lp_reg : int
88                 2 : L_2 Regularization is imposed
89                 1 : L_1 Regularization is imposed

```

```

90         0 : No regulariation is imposed
91
92     Returns:
93     -----
94     None
95     """
96     self.lp_reg = lp_reg
97
98     def predict(self, params, x):
99         """
100         Parameters:
101         -----
102         params : class[LinearParameters]
103         x : array_like
104
105         Returns:
106         -----
107         a : array_like
108         dg : array_like
109         """
110         z = params.forward(x)
111         a, dg = apply_activation(z, "sigmoid")
112         return a, dg
113
114     def cost_function(self, params, x, y, lambda_=0.01, eps=1e-8):
115         """
116         Parameters:
117         -----
118         params : class[LinearParameters]
119         x : array_like
120         y : array_like
121         lambda_ : float
122             Default : 0.01
123         eps : float
124             Default : 1e-8
125
126         Returns:
127         -----
128         cost : float
129         """
130         n = y.shape[1]
131
132         R = np.sum(np.abs(params.w) ** self.lp_reg)
133         R *= lambda_ / (2 * n)
134
135         a, _ = self.predict(params, x)
136         a = np.clip(a, eps, 1 - eps)

```

```

137
138         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
139
140         cost = float(np.squeeze(J + R))
141
142         return cost
143
144     def fit(self, x, y, learning_rate=0.1, lambda_=0.01, seed=1, num_iters=10000):
145         """
146         Parameters:
147         -----
148         x : array_like
149         y : array_like
150         learning_rate : float
151             Default : 0.1
152         lambda_ : float
153             Default : 0.0
154         num_iters : int
155             Default : 10000
156
157         Returns:
158         -----
159         costs : List[floats]
160         params : class[Parameters]
161         """
162         dims = (y.shape[0], x.shape[0])
163         n = x.shape[1]
164         params = LinearParameters(dims, True, seed)
165
166         if self.lp_reg == 0:
167             lambda_ = 0.0
168
169         costs = []
170         for i in range(num_iters):
171             a, _ = self.predict(params, x)
172             cost = self.cost_function(params, x, y, lambda_)
173             costs.append(cost)
174             dz = (a - y) / n
175             params.backward(dz, x)
176             params.update(learning_rate)
177
178             if i % 1000 == 0:
179                 print(f"Cost_after_iteration_{i}:_{cost}")
180
181         return params
182
183     def evaluate(self, params, x):

```

```

184         """
185         Parameters:
186         -----
187         params : class[Parameters]
188         x : array_like
189
190         Returns:
191         -----
192         y_hat : array_like
193         """
194         a, _ = self.predict(params, x)
195         y_hat = (~(a < 0.5)).astype(int)
196
197         return y_hat
198
199     def accuracy(self, params, x, y):
200         """
201         Parameters:
202         -----
203         params : class[Parameters]
204         x : array_like
205         y : array_like
206
207         Returns:
208         -----
209         accuracy : float
210         """
211         y_hat = self.evaluate(params, x)
212
213         accuracy = np.sum(y_hat == y) / y.shape[1]

```

1.3 Implementation in Python via **sklearn**

Here we include the general method of coding a logistic regression model via **scikit-learn**'s modeling library.

```

1  #! python3
2
3  import pandas as pd
4  import numpy as np
5  from sklearn.model_selection import train_test_split
6  from sklearn.linear_model import LogisticRegression
7
8  def main(csv):
9      df = pd.read_csv(csv)
10     dataset = df.values

```

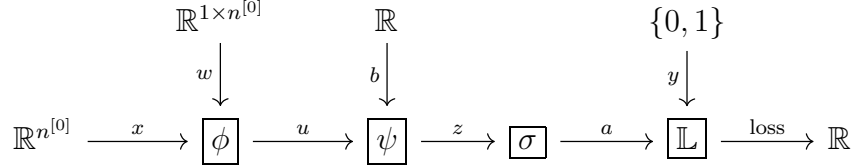
```

11     x = dataset[:, :10]
12     y = dataset[:, 10]
13
14     x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
15     mu = np.mean(x, axis=0, keepdims=True)
16     var = np.var(x, axis=0, keepdims=True)
17     x_train = (x_train - mu) / np.sqrt(var)
18     x_test = (x_test - mu) / np.sqrt(var)
19
20     log_reg = LogisticRegression()
21     log_reg.fit(x_train, y_train)
22     train_acc = log_reg.score(x_train, y_train)
23     print(f'The accuracy on the training set: {train_acc}.')
24     test_acc = log_reg.score(x_test, y_test)
25     print(f'The accuracy on the test set: {test_acc}.')

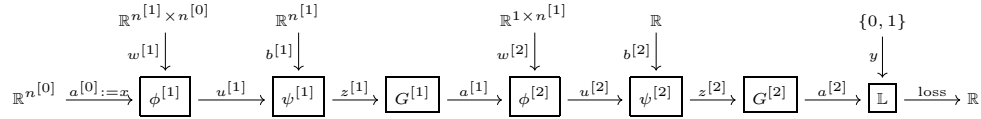
```

2 Neural Networks: A Single Hidden Layer

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



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:



In the above diagram, we use $\cdot^{[0]}$ to denote everything in layer-0, i.e., the input layer; we use $\cdot^{[1]}$ to denote everything in layer-1, i.e., the hidden layer; and we use $\cdot^{[2]}$ to denote everything in layer-2, i.e., the output layer. Moreover, we have the functions (where we suppress the layer-notation)

- $$\phi : \mathbb{R}^{n \times m} \times \mathbb{R}^m \rightarrow \mathbb{R}^n, \quad u := \phi(w, a) = wa,$$
- $$\psi : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n, \quad z := \psi(b, u) = u + b,$$
- $$G : \mathbb{R}^n \rightarrow \mathbb{R}^n, \quad a := G(z),$$

where G is the broadcasting of some activating function $g : \mathbb{R} \rightarrow \mathbb{R}$.

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

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

where $v \in \mathbb{R}^n$ and $\{e_i : 1 \leq i \leq n\}$ is the standard basis for \mathbb{R}^n . In practice, we will sometimes 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 $G : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is the broadcasting of g from \mathbb{R} to \mathbb{R}^n . Then the differential $dG_z : T_z \mathbb{R}^n \rightarrow T_{G(z)} \mathbb{R}^n$ is given by

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

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

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

We use the notation

$$G'(z) := [g'(z^i)] \in \mathbb{R}^n,$$

and thus may write

$$dG_z(v) = G'(z) \odot \xi.$$

Furthermore, we have that for $\zeta \in T_{G(z)} \mathbb{R}^n$,

$$rG_z(\zeta) = G'(z) \odot \zeta.$$

Proof: We calculate

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

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.

Furthermore, for $\zeta \in T_{G(z)}\mathbb{R}^n$, we have that

$$\begin{aligned}\langle rG_z(\zeta), \xi \rangle_{\mathbb{R}^n} &= \langle \zeta, dG_z(\xi) \rangle_{\mathbb{R}^n} \\ &= \langle \zeta, G'(z) \odot \xi \rangle_{\mathbb{R}^n} \\ &= \langle G'(z) \odot \zeta, \xi \rangle_{\mathbb{R}^n},\end{aligned}$$

and the result follows. \square

Returning to our network, we see call the full composition of network functions resulting in $a^{[2]}$, the *forward propagation*. That is, given an example $x \in \mathbb{R}^{n^{[0]}}$, we have that

$$a^{[2]} = G^{[2]}(\psi^{[2]}(b^{[2]}, \phi^{[2]}(w^{[2]}, G^{[1]}(\psi^{[1]}(b^{[1]}, \phi^{[1]}(w^{[1]}, x)))))).$$

2.1 Activation Functions

There are mainly only a handful of activating functions we consider for our non-linearity conditions (but many more built from these that follow).

2.1.1 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

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

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

2.1.2 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

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

Furthermore, we note that

$$\frac{1}{2} \left(\tanh\left(\frac{z}{2}\right) + 1 \right) = \sigma(z).$$

Indeed,

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

as desired.

2.1.3 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.

2.1.4 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 this function on the outer-layer to obtain a probability distribution over our predicted labels when dealing with multi-class regression. Let

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

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

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

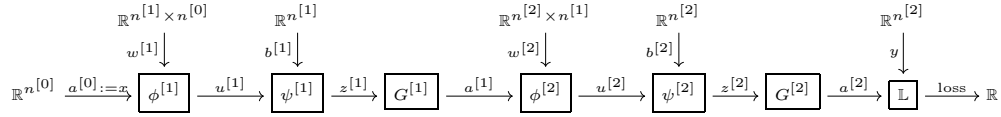
That is, as a map $dS_z : T_z \mathbb{R}^m \rightarrow T_{S(z)} \mathbb{R}^m$, we have that

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

and we make note that dS_z is symmetric (i.e., it's also the reverse differential).

2.2 Backward Propagation

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

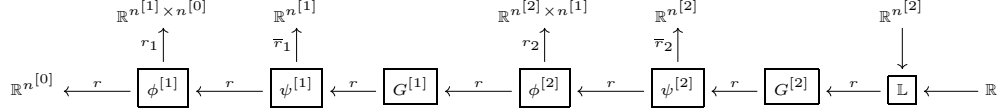
$$\psi^{[\ell]}(b, x) = x + b.$$

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

$$\phi(w, x) = wx.$$

Then we have two differentials to consider:

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

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

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

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

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

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

$$\begin{aligned} d_1\phi_{(w,x)}(\eta) &= \eta x \\ &= R_x(\eta); \end{aligned}$$

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

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

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

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

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:

•

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

•

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

•

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

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

$$\begin{aligned} w^{[1]} &:= w^{[1]} - r_1 \mathbb{J}_p(\alpha) \\ b^{[1]} &:= b^{[1]} - \bar{r}_1 \mathbb{J}_p(\alpha) \\ w^{[2]} &:= w^{[2]} - r_2 \mathbb{J}_p(\alpha) \\ b^{[2]} &:= b^{[2]} - \bar{r}_2 \mathbb{J}_p(\alpha). \end{aligned}$$

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

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

and hence

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

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

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

and hence

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

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

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

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

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

and hence

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

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

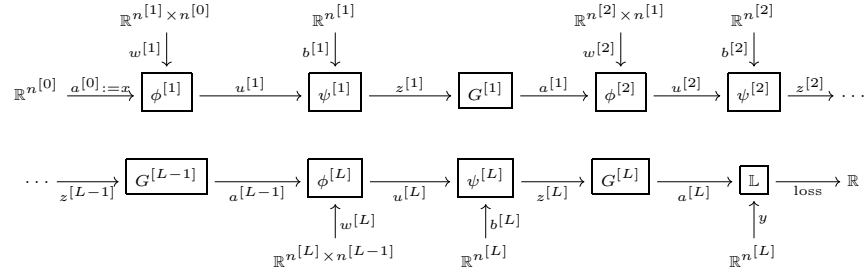
$$\begin{aligned} r_1 F_p &= r_1 (\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1 \phi_p^{[1]} \circ r\psi_{u^{[1]}}^{[1]} \circ rG_{z^{[1]}}^{[1]} \circ r\phi_{a^{[1]}}^{[2]} \circ r\psi_{u^{[2]}}^{[2]} \circ rG_{z^{[2]}}^{[2]}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

and hence

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

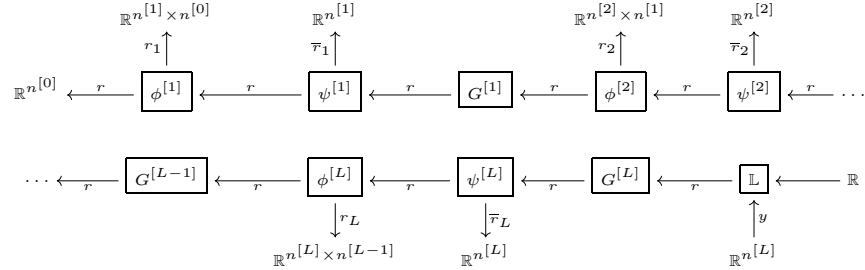
3 Deep Neural Networks

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



In general nothing fundamentally changes when adding more layers to a network. We may have different activator functions for each layer, but the general outline of computing forward propagation via composition, and then apply gradient descent by using reverse differentiation to “backtrack” through the network. Here we give a more general outline for computing our desired gradients.

To this end, we reverse our network to use reverse differentiation:



We compute differentials recursively as follows:

1. Define $\delta^{[L]}_j \in \mathbb{R}^{n^{[L]}}$ by

$$\begin{aligned} \delta^{[L]}_j &:= r(\mathbb{L} \circ G^{[L]})_{z^{[L]}_j} \\ &= rG^{[L]}_{z^{[L]}_j} \circ r\mathbb{L}_{(a^{[L]}_j, y_j)} \\ &= G^{[L]'}(z^{[L]}_j) \odot r\mathbb{L}_{(a^{[L]}_j, y_j)}. \end{aligned}$$

2. Compute

$$\frac{\partial \mathbb{J}}{\partial b^{[L]}} = \frac{1}{N} \sum_{j=1}^N \delta^{[L]}_j,$$

and

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial w^{[L]}} &= \frac{1}{N} \sum_{j=1}^N \delta^{[L]}_j a^{[L-1]T}_j \\ &= \frac{1}{N} \delta^{[L]} a^{[L-1]T}.\end{aligned}$$

3. Define $\delta^{[L-1]}_j \in \mathbb{R}^{n^{[L-1]}}$ by

$$\begin{aligned}\delta^{[L-1]}_j &:= r(\mathbb{L} \circ G^{[L]} \circ \psi^{[L]} \circ \phi^{[L]} \circ G^{[L-1]})_{z^{[L-1]}_j} \\ &= rG^{[L-1]}_{z^{[L-1]}_j} \circ r\phi^{[L]}_{(w^{[L]}, a^{[L-1]}_j)} \circ r\psi^{[L]}_{(b^{[L]}, u^{[L]}_j)} \circ rG^{[L]}_{z^{[L]}_j} \circ r\mathbb{L}_{(a^{[L]}_j, y_j)} \\ &= G^{[L-1]'}(z^{[L-1]}_j) \odot w^{[L]T} \cdot \delta^{[L]}_j.\end{aligned}$$

4. Compute

$$\frac{\partial \mathbb{J}}{\partial b^{[L-1]}} = \frac{1}{N} \sum_{j=1}^N \delta^{[L-1]}_j$$

and

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial w^{[L-1]}} &= \frac{1}{N} \sum_{j=1}^N \delta^{[L-1]}_j a^{[L-2]T}_j \\ &= \frac{1}{N} \delta^{[L-1]} a^{[L-2]T}.\end{aligned}$$

5. Given $\delta^{[\ell+1]}_j \in \mathbb{R}^{n^{[\ell+1]}}$, define $\delta^{[\ell]}_j \in \mathbb{R}^{n^{[\ell]}}$ by

$$\delta^{[\ell]}_j := G^{[\ell]'}(z^{[\ell]}_j) \odot w^{[\ell+1]T} \delta^{[\ell+1]}_j.$$

6. Compute

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

and

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

with the caveat that if $\ell = 1$, $a^{[0]} := x$, and we've completed the recursion.

3.1 Implementation in Python via numpy

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  #! python3
2
3  import numpy as np
4
5  from mllib.utils import LinearParameters, apply_activation
6
7
8  class NeuralNetwork:
9      def __init__(self, config):
10         """
11         Parameters:
12         -----
13         config : Dict
14             config['lp_reg'] = 0,1,2
15             config['nodes'] = List[int]
16             config['bias'] = List[Boolean]
17             config['activators'] = List[str]
18
19         Returns:
20         -----
21         None
22         """
23         self.config = config
24         self.lp_reg = config["lp_reg"]
25         self.nodes = config["nodes"]
26         self.bias = config["bias"]
27         self.activators = config["activators"]
28         self.L = len(config["nodes"]) - 1
29
30     def forward_propagation(self, params, x):
31         """
32         Parameters:
33         -----
34         params : Dict[class[Parameters]]
35             params[l].w = Weights
36             params[l].bias = Boolean
37             params[l].b = Bias
38         x : array_like
39
40         Returns:
```

```

41         -----
42         cache = Dict[array_like]
43             cache['a'] = a
44             cache['dg'] = dg
45
46         """
47         # Initialize dictionaries
48         a = {}
49         dg = {}
50
51         a[0], dg[0] = apply_activation(x, self.activators[0])
52
53         for l in range(1, self.L + 1):
54             z = params[l].forward(a[l - 1])
55             a[l], dg[l] = apply_activation(z, self.activators[l])
56
57         cache = {"a": a, "dg": dg}
58         return cache
59
60     def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
61         """
62         Parameters:
63         -----
64         params: class[Parameters]
65         a: array_like
66         y: array_like
67         lambda_: float
68             Default: 0.01
69         eps: float
70             Default: 1e-8
71
72         Returns:
73         -----
74         cost: float
75         """
76         n = y.shape[1]
77         if self.lp_reg == 0:
78             lambda_ = 0.0
79
80         # Compute regularization term
81         R = 0
82         for param in params.values():
83             R += np.sum(np.abs(param.w) ** self.lp_reg)
84         R *= lambda_ / (2 * n)
85
86         # Compute unregularized cost
87         a = np.clip(a, eps, 1 - eps) # Bound a for stability

```

```

88         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
89
90         cost = float(np.squeeze(J + R))
91
92         return cost
93
94     def backward_propagation(self, params, cache, y):
95         """
96         Parameters:
97         -----
98         params : Dict[class[Parameters]]
99             params[l].w = Weights
100             params[l].bias = Boolean
101             params[l].b = Bias
102         cache : Dict[array_like]
103             cache['a'] : array_like
104             cache['dg'] : array_like
105         y : array_like
106
107         Returns:
108         -----
109         None
110         """
111
112         # Retrieve cache
113         a = cache["a"]
114         dg = cache["dg"]
115
116         # Initialize differentials along the network
117         delta = {}
118         delta[self.L] = (a[self.L] - y) / y.shape[1]
119
120         for l in reversed(range(1, self.L + 1)):
121             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1])
122
123     def update_parameters(self, params, learning_rate):
124         """
125         Parameters:
126         -----
127         params : Dict[class[Parameters]]
128             params[l].w = Weights
129             params[l].b = Bias
130         learning_rate : float
131
132         Returns:
133         -----
134         None

```

```

135         """
136         for param in params.values():
137             param.update(learning_rate)
138
139     def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
140         """
141         Parameters:
142         -----
143         x : array_like
144         y : array_like
145         learning_rate : float
146             Default : 0.1
147         lambda_ : float
148             Default : 0.0
149         num_iters : int
150             Default : 10000
151
152         Returns:
153         -----
154         costs : List[floats]
155         params : class[Parameters]
156         """
157         # Initialize parameters per layer
158         params = {}
159         for l in range(1, self.L + 1):
160             params[l] = LinearParameters(
161                 (self.nodes[l], self.nodes[l - 1]), self.bias[l]
162             )
163
164         costs = []
165         for i in range(num_iters):
166             cache = self.forward_propagation(params, x)
167             cost = self.cost_function(params, cache["a"][self.L], y, lambda_)
168             costs.append(cost)
169             self.backward_propagation(params, cache, y)
170             self.update_parameters(params, learning_rate)
171
172             if i % 1000 == 0:
173                 print(f"Cost_after_iteration_{i}:_{cost}")
174
175         return params
176
177     def evaluate(self, params, x):
178         """
179         Parameters:
180         -----
181         params : class[Parameters]

```

```

182         x : array_like
183
184         Returns:
185         -----
186         y_hat : array_like
187         """
188         cache = self.forward_propagation(params, x)
189         a = cache["a"][self.L]
190         y_hat = (~(a < 0.5)).astype(int)
191         return y_hat
192
193     def accuracy(self, params, x, y):
194         """
195         Parameters:
196         -----
197         params : class[Parameters]
198         x : array_like
199         y : array_like
200
201         Returns:
202         -----
203         accuracy : float
204         """
205         y_hat = self.evaluate(params, x)
206         acc = np.sum(y_hat == y) / y.shape[1]
207
208         return acc

```

3.2 Implementation in Python via tensorflow

We implement a neural network using tensorflow.keras.

```

1  #! python3
2
3  import pandas as pd
4  import numpy as np
5  from sklearn.model_selection import train_test_split
6  from tensorflow import keras
7  from keras import Model, Input
8  from keras.layers import Dense
9
10 def keras_functional_nn(csv):
11     df = pd.read_csv(csv)
12     dataset = df.values
13     x, y = dataset[:, :-1], dataset[:, -1].reshape(-1, 1)
14     x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.15)

```

```

15 train = {'x' : x_train, 'y' : y_train}
16 test = {'x' : x_test, 'y' : y_test}
17 mu = np.mean(train['x'], axis=0, keepdims=True)
18 var = np.var(train['x'], axis=0, keepdims=True)
19 train['x'] = (train['x'] - mu) / np.sqrt(var)
20 test['x'] = (test['x'] - mu) / np.sqrt(var)
21
22 ## Define network structure
23 input_layer = Input(shape=(10,))
24 hidden_layer_1 = Dense(
25     32,
26     activation='relu',
27     kernel_initializer='he_normal',
28     bias_initializer='zeros'
29 )(input_layer)
30 hidden_layer_2 = Dense(
31     8,
32     activation='relu',
33     kernel_initializer='he_normal',
34     bias_initializer='zeros'
35 )(hidden_layer_1)
36 output_layer = Dense(
37     1,
38     activation='sigmoid',
39     kernel_initializer='he_normal',
40     bias_initializer='zeros'
41 )(hidden_layer_2)
42
43 model = Model(inputs=input_layer, outputs=output_layer)
44 model.summary()
45
46 ## Compile desired model
47 model.compile(
48     loss='binary_crossentropy',
49     optimizer='adam',
50     metrics=['accuracy']
51 )
52
53 ## Train the model
54 hist = model.fit(
55     train['x'],
56     train['y'],
57     batch_size=32,
58     epochs=150,
59     validation_split=0.17
60 )
61

```



```
62     ## Evaluate the model
63     test_scores = model.evaluate(test['x'], test['y'], verbose=2)
64     print(f'Test_Loss:_{test_scores[0]}')
65     print(f'Test_Accuracy:_{test_scores[1]}')
```

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 ## Classes
2
3 ## Shuffle, split and normalize full dataset
4 class ProcessData():
5     def __init__(self, x, y, test_percent, dev_percent=0.0, seed=101, shuffle=True,
6         """
7         Parameters:
8         -----
9         x : array_like
10             x.shape = (examples, features)
11         y : array_like
12             y.shape = (examples, labels)
13         test_percent : float
14         dev_percent : Tuple(floats)
15         seed : int
16             Default = 1
17         shuffle : Boolean
18             Default = True
19         feat_as_col : Boolean
20             Default = True
21
22         Returns:
23         -----
24         None
25         """
26         self.x = x
27         self.y = y
28         self.test_percent = test_percent
29         self.dev_percent = dev_percent
30         self.seed = seed
31         self.shuffle = shuffle
32         self.feat_as_col = feat_as_col
33
34         self.split()
35         self.normalize()
36
37         print(f"x_train.shape:_{self.train['x'].shape}")
38         print(f"y_train.shape:_{self.train['y'].shape}")
39         print(f"x_test.shape:_{self.test['x'].shape}")
40         print(f"y_test.shape:_{self.test['y'].shape}")
41         if self.dev_percent > 0.0:
42             print(f"x_dev.shape:_{self.dev['x'].shape}")
43             print(f"y_dev.shape:_{self.dev['y'].shape}")
44
```

```

45     def split(self):
46         """
47         Parameters:
48         -----
49         None
50
51         Returns:
52         -----
53         None
54         """
55         x_aux, x_test, y_aux, y_test = train_test_split(self.x, self.y, test_size=s
56         left_over = 1 - self.test_percent
57         aux_perc = self.dev_percent / left_over
58         x_train, x_dev, y_train, y_dev = train_test_split(x_aux, y_aux, test_size=a
59
60         if self.feats_as_col:
61             self.train = {'x' : x_train, 'y' : y_train}
62             self.test = {'x' : x_test, 'y' : y_test}
63             self.dev = {'x' : x_dev, 'y' : y_dev}
64         else:
65             self.train = {'x' : x_train.T, 'y' : y_train.T}
66             self.test = {'x' : x_test.T, 'y' : y_test.T}
67             self.dev = {'x' : x_dev.T, 'y' : y_dev.T}
68
69     def normalize(self, z=None, eps=0.0):
70         """
71         Parameters:
72         -----
73         z : array_like
74             Default : None - For initialization
75         eps : float
76             Default 0.0 - For stability
77
78         Returns:
79         z_scale : array_like
80         """
81         if z == None:
82             x = self.train['x']
83             axis = 0 if self.feats_as_col else 1
84             self.mu = np.mean(x, axis=axis, keepdims=True)
85             self.var = np.var(x, axis=axis, keepdims=True)
86             self.theta = 1 / np.sqrt(self.var + eps)
87             self.train['x'] = self.theta * (x - self.mu)
88             self.test['x'] = self.theta * (self.test['x'] - self.mu)
89             self.dev['x'] = self.theta * (self.dev['x'] - self.mu)
90
91         else:

```

```

92         z_scale = self.theta * (z - self.mu)
93         return z_scale
94
95 ## Shuffle and create mini-batches during training
96 class ShuffleBatchData():
97     def __init__(self, data, batch_size, seed=10101):
98         """
99         Parameters:
100         -----
101         data : Dict[array_like]

```

5 Regularization

Suppose we're training an L -layer neural network with dataset $\{(x_j, y_j)\} \subset \mathbb{R}^{n^{[0]}} \times \mathbb{R}^{n^{[L]}}$ with N examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{n^{[L]}} \times \mathbb{R}^{n^{[L]}} \rightarrow \mathbb{R}$, then we have our cost function \mathbb{J} defined on our one-parameter families of parameters w and b given by

$$\mathbb{J}(w, b) = \frac{1}{N} \sum_{j=1}^N \mathbb{L}(a^{[L]}_j, y_j).$$

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

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

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 tuning via 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

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

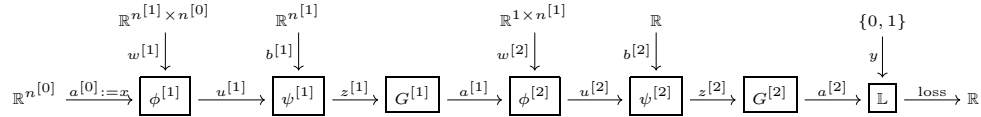
$$= \arg \min_{w, b} \left\{ \frac{1}{n_X} \sum_{(x, y) \in \mathbb{X}} \mathbb{L}(a^{[L]}, y) + \frac{\lambda}{2n_X} \sum_{\ell=1}^L \|w^{[\ell]}\|_F^2 \right\}$$

which is 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 (Inverted) Dropout Regularization

For illustrative purposes, suppose we have a 2-layer neural network of the following form:



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

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

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

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

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

where

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

- ii. During forward propagation, we redefine

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

- iii. During backward propagation, we define

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

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

5.1.1 Python Implementation

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

```

1 #! python3
2
3 import numpy as np
4
5 from mlLib.utils import LinearParameters, apply_activation
6
7
8 class NeuralNetwork():
9     def __init__(self, config):
10         """
11         Parameters:
12         -----
13         config : Dict
14             config['lp_reg'] = 0,1,2
15             config['nodes'] = List[int]
```

```

16         config['bias'] = List[Boolean]
17         config['activators'] = List[str]
18         config['keep_probs'] = List[float]
19
20     Returns:
21     -----
22     None
23     """
24     self.config = config
25     self.lp_reg = config['lp_reg']
26     self.nodes = config['nodes']
27     self.bias = config['bias']
28     self.activators = config['activators']
29     self.keep_probs = config['keep_probs']
30     self.L = len(config['nodes']) - 1
31
32     def init_dropout(self, num_examples, seed=1):
33         """
34         Parameters:
35         -----
36         num_examples : int
37         seed : int
38             Default: 1 # For reproducibility
39
40         Returns:
41         -----
42         D : Dict[layer : array_like]
43         """
44         np.random.seed(seed)
45         D = {}
46         for l in range(self.L + 1):
47             D[l] = np.random.rand(self.nodes[l], num_examples)
48             D[l] = (D[l] < self.keep_probs[l]).astype(int)
49             D[l] = D[l] / self.keep_probs[l]
50             assert (D[l].shape == (self.nodes[l], num_examples)), "Dropout_matrices."
51
52         return D
53
54     def forward_propagation(self, params, x, dropout=None):
55         """
56         Parameters:
57         -----
58         params : Dict[class[Parameters]]
59             params[l].w = Weights
60             params[l].bias = Boolean
61             params[l].b = Bias
62         x : array_like

```

```

63
64     Returns:
65     -----
66     cache = Dict[array_like]
67         cache['a'] = a
68         cache['dg'] = dg
69
70     """
71     # Initialize dictionaries
72     a = {}
73     dg = {}
74
75     a[0], dg[0] = apply_activation(x, self.activators[0])
76     if dropout != None:
77         a[0] = dropout[0] * a[0]
78
79     for l in range(1, self.L + 1):
80         z = params[l].forward(a[l - 1])
81         a[l], dg[l] = apply_activation(z, self.activators[l])
82         if dropout != None:
83             a[l] = dropout[l] * a[l]
84
85     cache = {'a': a, 'dg': dg}
86     return cache
87
88 def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
89     """
90     Parameters:
91     -----
92     params: class[Parameters]
93     a: array_like
94     y: array_like
95     lambda_: float
96         Default: 0.01
97     eps: float
98         Default: 1e-8
99
100     Returns:
101     -----
102     cost: float
103     """
104     n = y.shape[1]
105     if self.lp_reg == 0:
106         lambda_ = 0.0
107
108     # Compute regularization term
109     R = 0

```

```

110         for param in params.values():
111             R += np.sum(np.abs(param.w) ** self.lp_reg)
112         R *= (lambda_ / (2 * n))
113
114         # Compute unregularized cost
115         a = np.clip(a, eps, 1 - eps) # Bound a for stability
116         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
117
118         cost = float(np.squeeze(J + R))
119
120         return cost
121
122     def backward_propagation(self, params, cache, y, dropout):
123         """
124         Parameters:
125         -----
126         params : Dict[class[Parameters]]
127                 params[l].w = Weights
128                 params[l].bias = Boolean
129                 params[l].b = Bias
130         cache : Dict[array_like]
131                 cache['a'] : array_like
132                 cache['dg'] : array_like
133         y : array_like
134
135         Returns:
136         -----
137         None
138         """
139
140         # Retrieve cache
141         a = cache['a']
142         dg = cache['dg']
143
144         # Initialize differentials along the network
145         delta = {}
146         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
147
148         for l in reversed(range(1, self.L + 1)):
149             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l]
150
151     def update_parameters(self, params, learning_rate=0.1):
152         """
153         Parameters:
154         -----
155         params : Dict[class[Parameters]]
156                 params[l].w = Weights

```

```

157         params[l].bias = Boolean
158         params[l].b = Bias
159     learning_rate : float
160         Default : 0.01
161
162     Returns:
163     -----
164     None
165     """
166     for param in params.values():
167         param.update(learning_rate)
168
169 def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
170     """
171     Parameters:
172     -----
173     x : array_like
174     y : array_like
175     learning_rate : float
176         Default : 0.1
177     lambda_ : float
178         Default : 0.0
179     num_iters : int
180         Default : 10000
181
182     Returns:
183     -----
184     costs : List[floats]
185     params : class[Parameters]
186     """
187     # Initialize parameters per layer
188     params = {}
189     for l in range(1, self.L + 1):
190         params[l] = LinearParameters(
191             (self.nodes[l], self.nodes[l - 1]), self.bias[l])
192
193     costs = []
194     for i in range(num_iters):
195         dropout = self.init_dropout(x.shape[1])
196         cache = self.forward_propagation(params, x, dropout)
197         cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
198         costs.append(cost)
199         self.backward_propagation(params, cache, y, dropout)
200         self.update_parameters(params, learning_rate)
201
202         if i % 1000 == 0:
203             print(f'Cost_after_iteration_{i}:_{cost}')

```

```

204
205         return params
206
207     def evaluate(self, params, x):
208         """
209         Parameters:
210         -----
211         params : class[Parameters]
212         x : array_like
213
214         Returns:
215         -----
216         y_hat : array_like
217         """
218         cache = self.forward_propagation(params, x)
219         a = cache['a'][self.L]
220         y_hat = (~(a < 0.5)).astype(int)
221         return y_hat
222
223     def accuracy(self, params, x, y):
224         """
225         Parameters:
226         -----
227         params : class[Parameters]
228         x : array_like
229         y : array_like
230
231         Returns:
232         -----
233         accuracy : float
234         """
235         y_hat = self.evaluate(params, x)
236         acc = np.sum(y_hat == y) / y.shape[1]
237
238         return acc

```

5.2 Data Augmentation

This section requires work.

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

5.3 Early Stopping

This section requires work.

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

6 Gradients and Numerical Remarks

This section requires work. See “He Initialization” and “Xavier Initialization”

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

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

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

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 via numpy

This python function can check the reverse differential rf_x for the following types of functions f :

- $f : \mathbb{R} \rightarrow \mathbb{R}$
- $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and $f : \mathbb{R} \rightarrow \mathbb{R}^n$
- $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ and $f : \mathbb{R} \rightarrow \mathbb{R}^{m \times n}$
- $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^k$ and $f : \mathbb{R}^k \rightarrow \mathbb{R}^{m \times n}$
- $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^{k \times l}$

```
1  #! python3
2
3  import numpy as np
4  from numpy.linalg import norm
5
6  ## Checking the reverse differential of a function
7  def differential_check(f, x, eps=1e-3):
8      """
9      Parameters:
10     -----
11     f : function
12     x : array_like
13     eps : float
14         Default = 10^{-3}
15
16     Returns:
17     -----
18     error
19     """
20     y, rf = f(x)
21     x = np.array(x)
22     if len(x.shape) == 0:
23         x = x.reshape(1, 1)
24     elif len(x.shape) == 1:
25         x = x.reshape(-1, 1)
26     if len(y.shape) == 0:
27         y = y.reshape(1, 1)
28     elif len(y.shape) == 1:
```

```

29         y = y.reshape(-1, 1)
30
31     # k, l = y.shape
32     m, n = x.shape
33     # F = np.zeros((m, n, k, l))
34     F = np.zeros((*x.shape, *y.shape))
35     rf = rf.reshape(*x.shape, *y.shape)
36
37     for i in range(m):
38         for j in range(n):
39             e = np.zeros((m, n))
40             e[i, j] = 1
41             x_plus = x + eps * e
42             x_minus = x - eps * e
43             f_plus, _ = f(x_plus)
44             f_minus, _ = f(x_minus)
45             f_diff = f_plus - f_minus
46             f_diff = f_diff.reshape(*y.shape)
47             F[i, j] = f_diff
48
49     F = F / (2 * eps)
50
51     error = norm(F - rf) / (norm(F) + norm(rf))
52
53     return error
54
55
56 def sigmoid(x):
57     ## sigmoid:  $\hat{y}^n \in \mathbb{R}^n$  ##
58     # n = 1 is valid
59     x = np.array(x)
60     sigma = 1 / (1 + np.exp(-x))
61
62     dsigma = np.diagflat(sigma * (1 - sigma))
63     rsigma = dsigma.T
64     return sigma, rsigma
65
66
67 def foo(x):
68     ## f:  $\hat{y}^3 \in \mathbb{R}^2$  ##
69     ## f(x, y, z) = (xy, z^2) ##
70
71     y = np.zeros((2, 1))
72     y[0] = x[0] * x[1]
73     y[1] = x[2] ** 2
74
75     J = np.zeros((2, 3))

```

```

76     J[0, 0] = x[1]
77     J[0, 1] = x[0]
78     J[1, 2] = 2 * x[2]
79
80     R = np.einsum("ij->ji", J)
81     return y, R
82
83
84 def bar(x):
85     ## f:  $\mathbb{R}^m \rightarrow \mathbb{R}^m$  ##
86     ## f(x) = x@v
87     np.random.seed(1)
88     m, n = x.shape
89     v = np.random.randn(n)
90     f = np.einsum("ij,_j", x, v)
91
92     J = np.zeros((m, m, n))
93     for mu in range(m):
94         for i in range(m):
95             for j in range(n):
96                 if mu == i:
97                     J[mu, i, j] = v[j]
98
99     R = np.einsum("kij->ijk", J)
100    return f, R
101
102
103 def baz(x):
104     ## f:  $\mathbb{R}^m \rightarrow \mathbb{R}^m$  ##
105     ## f(x) = x * x # The Hadmard square
106     m, n = x.shape
107     f = np.einsum("ij,ij->ij", x, x)
108
109     J = np.zeros((m, n, m, n))
110     for mu in range(m):
111         for nu in range(n):
112             for i in range(m):
113                 for j in range(n):
114                     if (mu == i) and (nu == j):
115                         J[mu, nu, i, j] = 2 * x[i, j]
116
117     R = np.einsum("ijkl->klij", J)
118    return f, R

```

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 $\lceil \cdot \rceil$ 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 $B = \lceil \frac{N}{b} \rceil$, 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 :

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

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.

7.0.1 Python Implementation via numpy

We show here our implementation of dropout and L^2 -regularization utilizing mini-batch gradient descent in numpy.

```
1 #! python3
2
3 import numpy as np
4
5 from mlLib.utils import LinearParameters, apply_activation
6
7 class ShuffleBatchData():
8     def __init__(self, data, batch_size, seed=10101):
9         """
10         Parameters:
11         -----
12         data : Dict[array_like]
13             data['x'] : array_like
14             data['y'] : array_like
15         batch_size : int
16         seed : int
17             Default: 10101
18
19         Returns:
20         None
21         """
22         self.data = data
23         self.batch_size = batch_size
24         self.seed = seed
25         self.idx = np.arange(data['x'].shape[1])
26         self.__N = data['x'].shape[1]
27
28         np.random.seed(seed)
29
30     def get_batches(self):
31         """
32         Parameters:
33         -----
34         None
35
36         Returns:
37         -----
38         None
39         """
40         np.random.shuffle(self.idx)
41         x_shuffled = self.data['x'][:, self.idx]
42         y_shuffled = self.data['y'][:, self.idx]
```

```

43
44     B = int(np.ceil(self.__N / self.batch_size))
45
46     batches = []
47     for i in range(B):
48         x_aux = x_shuffled[:, (self.batch_size * i):(self.batch_size * (i + 1))]
49         y_aux = y_shuffled[:, (self.batch_size * i):(self.batch_size * (i + 1))]
50         batches.append({'x' : x_aux, 'y' : y_aux})
51
52     return batches
53
54 class NeuralNetwork():
55     def __init__(self, config):
56         """
57         Parameters:
58         -----
59         config : Dict
60             config['lp_reg'] = 0,1,2
61             config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
62             config['nodes'] = List[int]
63             config['bias'] = List[Boolean]
64             config['activators'] = List[str]
65             config['keep_probs'] = List[float]
66
67         Returns:
68         -----
69         None
70         """
71         self.config = config
72         self.lp_reg = config['lp_reg']
73         self.batch_size = config['batch_size']
74         self.nodes = config['nodes']
75         self.bias = config['bias']
76         self.activators = config['activators']
77         self.keep_probs = config['keep_probs']
78         self.L = len(config['nodes']) - 1
79
80     def init_dropout(self, num_examples, seed=101011):
81         """
82         Parameters:
83         -----
84         num_examples : int
85         seed : int
86             Default: 1 # For reproducibility
87
88         Returns:
89         -----

```



```

90         D : Dict[layer : array_like]
91         """
92         np.random.seed(seed)
93         D = {}
94         for l in range(self.L + 1):
95             D[l] = np.random.rand(self.nodes[l], num_examples)
96             D[l] = (D[l] < self.keep_probs[l]).astype(int)
97             D[l] = D[l] / self.keep_probs[l]
98             assert (D[l].shape == (self.nodes[l], num_examples)), "Dropout_matrices."
99
100         return D
101
102     def forward_propagation(self, params, x, dropout=None):
103         """
104         Parameters:
105         -----
106         params : Dict[class[Parameters]]
107             params[l].w = Weights
108             params[l].bias = Boolean
109             params[l].b = Bias
110         x : array_like
111
112         Returns:
113         -----
114         cache = Dict[array_like]
115             cache['a'] = a
116             cache['dg'] = dg
117
118         """
119         # Initialize dictionaries
120         a = {}
121         dg = {}
122
123         a[0], dg[0] = apply_activation(x, self.activators[0])
124         if dropout != None:
125             a[0] = dropout[0] * a[0]
126
127         for l in range(1, self.L + 1):
128             z = params[l].forward(a[l - 1])
129             a[l], dg[l] = apply_activation(z, self.activators[l])
130             if dropout != None:
131                 a[l] = dropout[l] * a[l]
132
133         cache = {'a': a, 'dg': dg}
134         return cache
135
136     def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):

```

```

137         """
138         Parameters:
139         -----
140         params: Dict[LinearParameters]
141         a: array_like
142         y: array_like
143         lambda_: float
144             Default: 0.01
145         eps: float
146             Default: 1e-8
147
148         Returns:
149         -----
150         cost: float
151         """
152         n = y.shape[1]
153         if self.lp_reg == 0:
154             lambda_ = 0.0
155
156         # Compute regularization term
157         R = 0
158         for param in params.values():
159             R += np.sum(np.abs(param.w) ** self.lp_reg)
160         R *= (lambda_ / (2 * n))
161
162         # Compute unregularized cost
163         a = np.clip(a, eps, 1 - eps) # Bound a for stability
164         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
165
166         cost = float(np.squeeze(J + R))
167
168         return cost
169
170     def backward_propagation(self, params, cache, y, dropout):
171         """
172         Parameters:
173         -----
174         params : Dict[LinearParameters]
175             params[1].w = Weights
176             params[1].bias = Boolean
177             params[1].b = Bias
178         cache : Dict[array_like]
179             cache['a'] : array_like
180             cache['dg'] : array_like
181         y : array_like
182
183         Returns:

```

```

184         -----
185         None
186         """
187
188         # Retrieve cache
189         a = cache['a']
190         dg = cache['dg']
191
192         # Initialize differentials along the network
193         delta = {}
194         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
195
196         for l in reversed(range(1, self.L + 1)):
197             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l]
198
199     def update_parameters(self, params, learning_rate=0.1):
200         """
201         Parameters:
202         -----
203         params : Dict[LinearParameters]
204             params[l].w = Weights
205             params[l].bias = Boolean
206             params[l].b = Bias
207         learning_rate : float
208             Default : 0.01
209
210         Returns:
211         -----
212         None
213         """
214         for param in params.values():
215             param.update(learning_rate)
216
217     def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000):
218         """
219         Parameters:
220         -----
221         data : Dict[array_like]
222             data['x'] : array_like
223             data['y'] : array_like
224         learning_rate : float
225             Default : 0.1
226         lambda_ : float
227             Default : 0.0
228         num_iters : int
229             Default : 10000
230

```

```

231     Returns:
232     -----
233     costs : List[floats]
234     params : class[LinearParameters]
235     """
236     # Initialize parameters per layer
237     params = {}
238     for l in range(1, self.L + 1):
239         params[l] = LinearParameters(
240             (self.nodes[l], self.nodes[l - 1]), self.bias[l])
241
242     # Initialize batching
243     batching = ShuffleBatchData(data, self.batch_size)
244
245     costs = []
246     for i in range(num_iters):
247         batches = batching.get_batches()
248         for batch in batches:
249             x = batch['x']
250             y = batch['y']
251             dropout = self.init_dropout(x.shape[1])
252             cache = self.forward_propagation(params, x, dropout)
253             cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
254             costs.append(cost)
255             self.backward_propagation(params, cache, y, dropout)
256             self.update_parameters(params, learning_rate)
257
258             if i % 100 == 0:
259                 print(f'Cost_after_iteration_{i}:_{cost}')
260
261     return params
262
263 def evaluate(self, params, x):
264     """
265     Parameters:
266     -----
267     params : Dict[LinearParameters]
268     x : array_like
269
270     Returns:
271     -----
272     y_hat : array_like
273     """
274     cache = self.forward_propagation(params, x)
275     a = cache['a'][self.L]
276     y_hat = (~(a < 0.5)).astype(int)
277     return y_hat

```

```

278
279     def accuracy(self, params, data):
280         """
281         Parameters:
282         -----
283         params : Dict[LinearParameters]
284         data : Dict[array_like]
285             data['x'] : array_like
286             data['y'] : array_like
287
288         Returns:
289         -----
290         accuracy : float
291         """
292         x = data['x']
293         y = data['y']
294
295         y_hat = self.evaluate(params, x)
296         acc = np.sum(y_hat == y) / y.shape[1]
297
298         return acc

```

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

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

- iv. We update parameters

$$\begin{aligned} w^{\{t\}} &= w^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}^{\{t\}}}{\partial w} \\ b^{\{t\}} &= b^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}^{\{t\}}}{\partial b} \end{aligned}$$

Using this formulation of gradient descent, we insert EMA applied to the sequences of gradients depending on the iteration $t := iB + k$. 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

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

- 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

$$w^{\{t\}} = w^{\{t-1\}} - \alpha v_w^{\{t\}}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha v_b^{\{t\}}$$

7.2.1 Python Implementation via `numpy`

Here we build on our previous mini-batch implementation by optimizing via gradient descent with momentum, implemented with the `numpy` package.

```

1  #! python3
2
3  import numpy as np
4
5  from mllib.utils import LinearParameters, ShuffleBatchData, apply_activation
6
7
8  class Momentum:
9      def __init__(self, param, bias, beta1=0.9):
10         """
11         Parameters:
12         -----
13         param : LinearParameters
14         bias : Bool
15         beta1 : float
16             Default = 0.9
17
18         Returns:
19         -----
20         None

```

```

21         """
22         self.bias = bias
23         self.beta1 = beta1
24         self.w = np.zeros(param.w.shape)
25         if self.bias:
26             self.b = np.zeros(param.b.shape)
27
28     def update(self, param, learning_rate, iter, update_params=True):
29         """
30         Parameters:
31         -----
32         param : LinearParameter
33         learning_rate : float
34         iter : int
35         update_params : Bool
36             Default = True - Dictates return type
37
38         Returns:
39         -----
40         None OR v : Dict[array_like]
41         """
42         self.w = self.beta1 * self.w + (1 - self.beta1) * param.dw
43         vw_corrected = self.w / (1 - self.beta1**iter)
44         if update_params:
45             param.w = param.w - learning_rate * vw_corrected
46         if self.bias:
47             self.b = self.beta1 * self.b + (1 - self.beta1) * param.db
48             vb_corrected = self.b / (1 - self.beta1**iter)
49             if update_params:
50                 param.b = param.b - learning_rate * vb_corrected
51         if not update_params:
52             v = {}
53             v["w"] = vw_corrected
54             if self.bias:
55                 v["b"] = vb_corrected
56             return v
57
58
59 class NeuralNetwork:
60     def __init__(self, config):
61         """
62         Parameters:
63         -----
64         config : Dict
65             config['lp_reg'] = 0,1,2
66             config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
67             config['nodes'] = List[int]

```

```

68         config['bias'] = List[Boolean]
69         config['activators'] = List[str]
70         config['keep_probs'] = List[float]
71
72     Returns:
73     -----
74     None
75     """
76     self.config = config
77     self.lp_reg = config["lp_reg"]
78     self.batch_size = config["batch_size"]
79     self.nodes = config["nodes"]
80     self.bias = config["bias"]
81     self.activators = config["activators"]
82     self.keep_probs = config["keep_probs"]
83     self.L = len(config["nodes"]) - 1
84
85     def init_dropout(self, num_examples, seed=101011):
86         """
87         Parameters:
88         -----
89         num_examples : int
90         seed : int
91             Default: 1 # For reproducibility
92
93         Returns:
94         -----
95         D : Dict[layer : array_like]
96         """
97         np.random.seed(seed)
98         D = {}
99         for l in range(self.L + 1):
100             D[l] = np.random.rand(self.nodes[l], num_examples)
101             D[l] = (D[l] < self.keep_probs[l]).astype(int)
102             D[l] = D[l] / self.keep_probs[l]
103             assert D[l].shape == (
104                 self.nodes[l],
105                 num_examples,
106             ), "Dropout_matrices_are_the_wrong_shape"
107
108         return D
109
110     def forward_propagation(self, params, x, dropout=None):
111         """
112         Parameters:
113         -----
114         params : Dict[class[Parameters]]

```

```

115         params[l].w = Weights
116         params[l].bias = Boolean
117         params[l].b = Bias
118     x : array_like
119
120     Returns:
121     -----
122     cache = Dict[array_like]
123         cache['a'] = a
124         cache['dg'] = dg
125
126     """
127     # Initialize dictionaries
128     a = {}
129     dg = {}
130
131     a[0], dg[0] = apply_activation(x, self.activators[0])
132     if dropout != None:
133         a[0] = dropout[0] * a[0]
134
135     for l in range(1, self.L + 1):
136         z = params[l].forward(a[l - 1])
137         a[l], dg[l] = apply_activation(z, self.activators[l])
138         if dropout != None:
139             a[l] = dropout[l] * a[l]
140
141     cache = {"a": a, "dg": dg}
142     return cache
143
144 def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
145     """
146     Parameters:
147     -----
148     params: Dict[LinearParameters]
149     a: array_like
150     y: array_like
151     lambda_: float
152         Default: 0.01
153     eps: float
154         Default: 1e-8
155
156     Returns:
157     -----
158     cost: float
159     """
160     n = y.shape[1]
161     if self.lp_reg == 0:

```

```

162         lambda_ = 0.0
163
164     # Compute regularization term
165     R = 0
166     for param in params.values():
167         R += np.sum(np.abs(param.w) ** self.lp_reg)
168     R *= lambda_ / (2 * n)
169
170     # Compute unregularized cost
171     a = np.clip(a, eps, 1 - eps) # Bound a for stability
172     J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
173
174     cost = float(np.squeeze(J + R))
175
176     return cost
177
178 def backward_propagation(self, params, cache, y, dropout):
179     """
180     Parameters:
181     -----
182     params : Dict[LinearParameters]
183             params[l].w = Weights
184             params[l].bias = Boolean
185             params[l].b = Bias
186     cache : Dict[array_like]
187             cache['a'] : array_like
188             cache['dg'] : array_like
189     y : array_like
190
191     Returns:
192     -----
193     None
194     """
195
196     # Retrieve cache
197     a = cache["a"]
198     dg = cache["dg"]
199
200     # Initialize differentials along the network
201     delta = {}
202     delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
203
204     for l in reversed(range(1, self.L + 1)):
205         delta[l - 1] = (
206             dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
207         )
208

```

```

209     def update_parameters(self, params, moms, learning_rate, iter):
210         """
211         Parameters:
212         -----
213         params : Dict[LinearParameters]
214             params[l].w = Weights
215             params[l].b = Bias
216         moms : Dict[Momentum]
217         learning_rate : float
218         iter : int
219
220         Returns:
221         -----
222         None
223         """
224         for l in params.keys():
225             moms[l].update(params[l], learning_rate, iter, True)
226
227     def fit(
228         self,
229         data,
230         learning_rate=0.1,
231         lambda_=0.01,
232         num_epochs=10000,
233         print_cost_iter=1000,
234     ):
235         """
236         Parameters:
237         -----
238         data : Dict[array_like]
239             data['x'] : array_like
240             data['y'] : array_like
241         learning_rate : float
242             Default : 0.1
243         lambda_ : float
244             Default : 0.0
245         num_iters : int
246             Default : 10000
247         print_cost_iter : int
248             Default: 1000    # 0 Doesn't print costs
249
250         Returns:
251         -----
252         costs : List[floats]
253         params : class[LinearParameters]
254         """
255         # Initialize parameters and optimizer per layer

```

```

256     params = {}
257     moms = {}
258     for l in range(1, self.L + 1):
259         params[l] = LinearParameters(
260             (self.nodes[l], self.nodes[l - 1]), self.bias[l]
261         )
262         moms[l] = Momentum(params[l], self.bias[l])
263
264     # Initialize batching
265     batching = ShuffleBatchData(data, self.batch_size)
266
267     costs = []
268     for epoch in range(num_epochs):
269         batches = batching.get_batches()
270         B = len(batches)
271         k = 1
272         cost = 0
273         for batch in batches:
274             iter = (epoch * B) + k
275             x = batch["x"]
276             y = batch["y"]
277             dropout = self.init_dropout(x.shape[1])
278             cache = self.forward_propagation(params, x, dropout)
279             batch_cost = self.cost_function(params, cache["a"][self.L], y, lambda)
280             cost += x.shape[1] * batch_cost
281             self.backward_propagation(params, cache, y, dropout)
282             self.update_parameters(params, moms, learning_rate, iter)
283             k += 1
284         cost /= data["x"].shape[1]
285         costs.append(cost)
286
287         if (print_cost_iter != 0) and (epoch % print_cost_iter == 0):
288             print(f"Cost_after_epoch_{epoch}:_{cost}")
289
290     return params, costs
291
292 def evaluate(self, params, x):
293     """
294     Parameters:
295     -----
296     params : Dict[LinearParameters]
297     x : array_like
298
299     Returns:
300     -----
301     y_hat : array_like
302     """

```

```

303         cache = self.forward_propagation(params, x)
304         a = cache["a"][self.L]
305         y_hat = (~a < 0.5).astype(int)
306         return y_hat
307
308     def accuracy(self, params, data):
309         """
310         Parameters:
311         -----
312         params : Dict[LinearParameters]
313         data : Dict[array_like]
314             data['x'] : array_like
315             data['y'] : array_like
316
317         Returns:
318         -----
319         accuracy : float
320         """
321         x = data["x"]
322         y = data["y"]
323
324         y_hat = self.evaluate(params, x)
325         acc = np.sum(y_hat == y) / y.shape[1]
326
327         return acc

```

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 having 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 the update 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 perturbation). 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

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

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

7.3.1 Python Implementation via numpy

Here we implement the RMS Propagation algorithm using the numpy library.

```
1  #! python3
2
3  import numpy as np
4
5  from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation
6
7
8  class RMSProp:
9      def __init__(self, param, bias, beta2=0.9, eps=1e-8):
10         """
11         Parameters:
12         -----
13         params : LinearParameters
14         bias : Bool
15         beta2 : float
16             Default = 0.9
17         eps : float
18             Default = 10^{-8}
19
20         Returns:
21         None
22         """
23         self.bias = bias
24         self.beta2 = beta2
25         self.eps = eps
26         self.w = np.zeros(param.w.shape)
27         if self.bias:
28             self.b = np.zeros(param.b.shape)
29
30     def update(self, param, learning_rate, iter, update_params=True):
31         """
32         Parameters:
33         -----
34         params : LinearParameters
35         learning_rate : float
36         iter : int
37         update_params : Boolean
38             Default = True
39
40         Returns:
41         None OR v : Dict[array_like]
42         """
43         self.w = self.beta2 * self.w + (1 - self.beta2) * (param.dw**2)
44         sw_corrected = self.w / (1 - self.beta2**iter)
```

```

45         if update_params:
46             param.w = param.w - learning_rate * (
47                 param.dw / (np.sqrt(sw_corrected) + self.eps)
48             )
49         if self.bias:
50             self.b = self.beta2 * self.b + (1 - self.beta2) * (param.db**2)
51             sb_corrected = self.b / (1 - self.beta2**iter)
52             if update_params:
53                 param.b = param.b - learning_rate * (
54                     param.db / (np.sqrt(sb_corrected) + self.eps)
55                 )
56         if not update_params:
57             s = {}
58             s["w"] = sw_corrected
59             if self.bias:
60                 s["b"] = sb_corrected
61             return s
62
63
64     class NeuralNetwork:
65         def __init__(self, config):
66             """
67             Parameters:
68             -----
69             config : Dict
70                 config['lp_reg'] = 0,1,2
71                 config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
72                 config['nodes'] = List[int]
73                 config['bias'] = List[Boolean]
74                 config['activators'] = List[str]
75                 config['keep_probs'] = List[float]
76
77             Returns:
78             -----
79             None
80             """
81             self.config = config
82             self.lp_reg = config["lp_reg"]
83             self.batch_size = config["batch_size"]
84             self.nodes = config["nodes"]
85             self.bias = config["bias"]
86             self.activators = config["activators"]
87             self.keep_probs = config["keep_probs"]
88             self.L = len(config["nodes"]) - 1
89
90         def init_dropout(self, num_examples, seed=101011):
91             """

```

```

92     Parameters:
93     -----
94     num_examples : int
95     seed : int
96         Default: 1 # For reproducibility
97
98     Returns:
99     -----
100     D : Dict[layer : array_like]
101     """
102     np.random.seed(seed)
103     D = {}
104     for l in range(self.L + 1):
105         D[l] = np.random.rand(self.nodes[l], num_examples)
106         D[l] = (D[l] < self.keep_probs[l]).astype(int)
107         D[l] = D[l] / self.keep_probs[l]
108         assert D[l].shape == (
109             self.nodes[l],
110             num_examples,
111         ), "Dropout_matrices_are_the_wrong_shape"
112
113     return D
114
115 def forward_propagation(self, params, x, dropout=None):
116     """
117     Parameters:
118     -----
119     params : Dict[class[Parameters]]
120         params[l].w = Weights
121         params[l].bias = Boolean
122         params[l].b = Bias
123     x : array_like
124
125     Returns:
126     -----
127     cache = Dict[array_like]
128         cache['a'] = a
129         cache['dg'] = dg
130
131     """
132     # Initialize dictionaries
133     a = {}
134     dg = {}
135
136     a[0], dg[0] = apply_activation(x, self.activators[0])
137     if dropout != None:
138         a[0] = dropout[0] * a[0]

```

```

139
140     for l in range(1, self.L + 1):
141         z = params[l].forward(a[l - 1])
142         a[l], dg[l] = apply_activation(z, self.activators[l])
143         if dropout != None:
144             a[l] = dropout[l] * a[l]
145
146     cache = {"a": a, "dg": dg}
147     return cache
148
149 def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
150     """
151     Parameters:
152     -----
153     params: Dict[LinearParameters]
154     a: array_like
155     y: array_like
156     lambda_: float
157         Default: 0.01
158     eps: float
159         Default: 1e-8
160
161     Returns:
162     -----
163     cost: float
164     """
165     n = y.shape[1]
166     if self.lp_reg == 0:
167         lambda_ = 0.0
168
169     # Compute regularization term
170     R = 0
171     for param in params.values():
172         R += np.sum(np.abs(param.w) ** self.lp_reg)
173     R *= lambda_ / (2 * n)
174
175     # Compute unregularized cost
176     a = np.clip(a, eps, 1 - eps) # Bound a for stability
177     J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
178
179     cost = float(np.squeeze(J + R))
180
181     return cost
182
183 def backward_propagation(self, params, cache, y, dropout):
184     """
185     Parameters:

```

```

186         -----
187         params : Dict[LinearParameters]
188             params[l].w = Weights
189             params[l].bias = Boolean
190             params[l].b = Bias
191         cache : Dict[array_like]
192             cache['a'] : array_like
193             cache['dg'] : array_like
194         y : array_like
195
196         Returns:
197         -----
198         None
199         """
200
201         # Retrieve cache
202         a = cache["a"]
203         dg = cache["dg"]
204
205         # Initialize differentials along the network
206         delta = {}
207         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
208
209         for l in reversed(range(1, self.L + 1)):
210             delta[l - 1] = (
211                 dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
212             )
213
214     def update_parameters(self, params, rmsprops, learning_rate, iter):
215         """
216         Parameters:
217         -----
218         params : Dict[LinearParameters]
219             params[l].w = Weights
220             params[l].b = Bias
221         rmsprops : Dict[RMSProp]
222         learning_rate : float
223         iter : int
224
225         Returns:
226         -----
227         None
228         """
229         for l in params.keys():
230             rmsprops[l].update(params[l], learning_rate, iter, True)
231
232     def fit(

```

```

233         self,
234         data,
235         learning_rate=0.1,
236         lambda_=0.01,
237         num_epochs=10000,
238         print_cost_iter=1000,
239     ):
240         """
241         Parameters:
242         -----
243         data : Dict[array_like]
244             data['x'] : array_like
245             data['y'] : array_like
246         learning_rate : float
247             Default : 0.1
248         lambda_ : float
249             Default : 0.0
250         num_epochs : int
251             Default : 10000
252         print_cost_iter : int
253             Default: 1000    # 0 Doesn't print costs
254
255         Returns:
256         -----
257         costs : List[floats]
258         params : class[LinearParameters]
259         """
260         # Initialize parameters and optimizer per layer
261         params = {}
262         rmsprops = {}
263         for l in range(1, self.L + 1):
264             params[l] = LinearParameters(
265                 (self.nodes[l], self.nodes[l - 1]), self.bias[l]
266             )
267             rmsprops[l] = RMSProp(params[l], self.bias[l])
268
269         # Initialize batching
270         batching = ShuffleBatchData(data, self.batch_size)
271
272         costs = []
273         for epoch in range(num_epochs):
274             batches = batching.get_batches()
275             B = len(batches)
276             k = 1
277             cost = 0
278             for batch in batches:
279                 iter = (epoch * B) + k

```

```

280         x = batch["x"]
281         y = batch["y"]
282         dropout = self.init_dropout(x.shape[1])
283         cache = self.forward_propagation(params, x, dropout)
284         batch_cost = self.cost_function(params, cache["a"][self.L], y, lambda)
285         cost += x.shape[1] * batch_cost
286         self.backward_propagation(params, cache, y, dropout)
287         self.update_parameters(params, rmsprops, learning_rate, iter)
288         k += 1
289     cost /= data["x"].shape[1]
290     costs.append(cost)
291
292     if (print_cost_iter != 0) and (epoch % print_cost_iter == 0):
293         print(f"Cost_after_epoch_{epoch}:_{cost}")
294
295     return params, costs
296
297 def evaluate(self, params, x):
298     """
299     Parameters:
300     -----
301     params : Dict[LinearParameters]
302     x : array_like
303
304     Returns:
305     -----
306     y_hat : array_like
307     """
308     cache = self.forward_propagation(params, x)
309     a = cache["a"][self.L]
310     y_hat = (~(a < 0.5)).astype(int)
311     return y_hat
312
313 def accuracy(self, params, data):
314     """
315     Parameters:
316     -----
317     params : Dict[LinearParameters]
318     data : Dict[array_like]
319           data['x'] : array_like
320           data['y'] : array_like
321
322     Returns:
323     -----
324     accuracy : float
325     """
326     x = data["x"]

```



```

327         y = data["y"]
328
329         y_hat = self.evaluate(params, x)
330         acc = np.sum(y_hat == y) / y.shape[1]
331
332     return acc

```

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

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

- iv. Define

$$v_w^{\{t\}} = \beta_1 v_w^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}^{\{t\}}}{\partial w},$$

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

and 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. Utilize bias correction via:

$$\hat{v}_w^{\{t\}} = \frac{v_w^{\{t\}}}{1 - \beta_1^t}$$

$$\hat{v}_b^{\{t\}} = \frac{v_b^{\{t\}}}{1 - \beta_1^t}$$

$$\hat{s}_w^{\{t\}} = \frac{s_w^{\{t\}}}{1 - \beta_2^t}$$

$$\hat{s}_b^{\{t\}} = \frac{s_b^{\{t\}}}{1 - \beta_2^t}$$

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

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

7.4.1 Python Implementation via numpy

```

1 #! python3
2
3 import numpy as np
4
5 from mllib.utils import LinearParameters, ShuffleBatchData, Momentum, RMSProp
6 from mllib.utils import apply_activation
7
```

```

8
9 class Adam:
10     def __init__(self, param, bias, beta1=0.9, beta2=0.999, eps=1e-8):
11         """
12         Parameters:
13         -----
14         param : LinearParameters
15         bias : Bool
16         beta1 : float
17             Default = 0.9
18         beta2 : float
19             Default = 0.999
20         eps : float
21             Default = 10^{-8}
22
23         Returns:
24         None
25         """
26         self.bias = bias
27         self.beta1 = beta1
28         self.beta2 = beta2
29         self.eps = eps
30
31         self.mom = Momentum(param, self.bias, self.beta1)
32         self.rmsprop = RMSProp(param, self.bias, self.beta2, self.eps)
33
34     def update(self, param, learning_rate, iter):
35         """
36         Parameters:
37         -----
38         params : LinearParameters
39         learning_rate : float
40         iter : int
41
42         Returns:
43         None
44         """
45         v = self.mom.update(param, learning_rate, iter, False)
46         s = self.rmsprop.update(param, learning_rate, iter, False)
47
48         param.w = param.w - learning_rate * v["w"] / (np.sqrt(s["w"]) + self.eps)
49         if self.bias:
50             param.b = param.b - learning_rate * v["b"] / (np.sqrt(s["b"]) + self.eps)
51
52
53 class NeuralNetwork:
54     def __init__(self, config):

```

```

55     """
56     Parameters:
57     -----
58     config : Dict
59         config['lp_reg'] = 0,1,2
60         config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
61         config['nodes'] = List[int]
62         config['bias'] = List[Boolean]
63         config['activators'] = List[str]
64         config['keep_probs'] = List[float]
65
66     Returns:
67     -----
68     None
69     """
70     self.config = config
71     self.lp_reg = config["lp_reg"]
72     self.batch_size = config["batch_size"]
73     self.nodes = config["nodes"]
74     self.bias = config["bias"]
75     self.activators = config["activators"]
76     self.keep_probs = config["keep_probs"]
77     self.L = len(config["nodes"]) - 1
78
79     def init_dropout(self, num_examples, seed=101011):
80         """
81         Parameters:
82         -----
83         num_examples : int
84         seed : int
85             Default: 1 # For reproducibility
86
87         Returns:
88         -----
89         D : Dict[layer : array_like]
90         """
91         np.random.seed(seed)
92         D = {}
93         for l in range(self.L + 1):
94             D[l] = np.random.rand(self.nodes[l], num_examples)
95             D[l] = (D[l] < self.keep_probs[l]).astype(int)
96             D[l] = D[l] / self.keep_probs[l]
97             assert D[l].shape == (
98                 self.nodes[l],
99                 num_examples,
100             ), "Dropout_matrices_are_the_wrong_shape"
101

```

```

102         return D
103
104     def forward_propagation(self, params, x, dropout=None):
105         """
106         Parameters:
107         -----
108         params : Dict[class[Parameters]]
109                 params[1].w = Weights
110                 params[1].bias = Boolean
111                 params[1].b = Bias
112         x : array_like
113
114         Returns:
115         -----
116         cache = Dict[array_like]
117                 cache['a'] = a
118                 cache['dg'] = dg
119
120         """
121         # Initialize dictionaries
122         a = {}
123         dg = {}
124
125         a[0], dg[0] = apply_activation(x, self.activators[0])
126         if dropout != None:
127             a[0] = dropout[0] * a[0]
128
129         for l in range(1, self.L + 1):
130             z = params[l].forward(a[l - 1])
131             a[l], dg[l] = apply_activation(z, self.activators[l])
132             if dropout != None:
133                 a[l] = dropout[l] * a[l]
134
135         cache = {"a": a, "dg": dg}
136         return cache
137
138     def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
139         """
140         Parameters:
141         -----
142         params: Dict[LinearParameters]
143         a: array_like
144         y: array_like
145         lambda_: float
146                 Default: 0.01
147         eps: float
148                 Default: 1e-8

```

```

149
150     Returns:
151     -----
152     cost: float
153     """
154     n = y.shape[1]
155     if self.lp_reg == 0:
156         lambda_ = 0.0
157
158     # Compute regularization term
159     R = 0
160     for param in params.values():
161         R += np.sum(np.abs(param.w) ** self.lp_reg)
162     R *= lambda_ / (2 * n)
163
164     # Compute unregularized cost
165     a = np.clip(a, eps, 1 - eps) # Bound a for stability
166     J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
167
168     cost = float(np.squeeze(J + R))
169
170     return cost
171
172 def backward_propagation(self, params, cache, y, dropout):
173     """
174     Parameters:
175     -----
176     params : Dict[LinearParameters]
177             params[l].w = Weights
178             params[l].bias = Boolean
179             params[l].b = Bias
180     cache : Dict[array_like]
181            cache['a'] : array_like
182            cache['dg'] : array_like
183     y : array_like
184
185     Returns:
186     -----
187     None
188     """
189
190     # Retrieve cache
191     a = cache["a"]
192     dg = cache["dg"]
193
194     # Initialize differentials along the network
195     delta = {}

```

```

196         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
197
198     for l in reversed(range(1, self.L + 1)):
199         delta[l - 1] = (
200             dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
201         )
202
203     def update_parameters(self, params, adams, learning_rate, iter):
204         """
205         Parameters:
206         -----
207         params : Dict[LinearParameters]
208             params[l].w = Weights
209             params[l].b = Bias
210         adams : Dict[Adam]
211         learning_rate : float
212         iter : int
213
214         Returns:
215         -----
216         None
217         """
218         for l in params.keys():
219             adams[l].update(params[l], learning_rate, iter)
220
221     def fit(
222         self,
223         data,
224         learning_rate=0.1,
225         lambda_=0.01,
226         num_epochs=10000,
227         print_cost_iter=1000,
228     ):
229         """
230         Parameters:
231         -----
232         data : Dict[array_like]
233             data['x'] : array_like
234             data['y'] : array_like
235         learning_rate : float
236             Default : 0.1
237         lambda_ : float
238             Default : 0.01
239         num_epochs : int
240             Default : 10000
241         print_cost_iter : int
242             Default: 1000 # 0 Doesn't print costs

```

```

243
244 Returns:
245 -----
246 costs : List[floats]
247 params : Dict[LinearParameters]
248 """
249 # Initialize parameters and optimizer per layer
250 params = {}
251 adams = {}
252 for l in range(1, self.L + 1):
253     params[l] = LinearParameters(
254         (self.nodes[l], self.nodes[l - 1]), self.bias[l]
255     )
256     adams[l] = Adam(params[l], self.bias[l])
257
258 # Initialize batching
259 batching = ShuffleBatchData(data, self.batch_size)
260
261 costs = []
262 for epoch in range(num_epochs):
263     batches = batching.get_batches()
264     B = len(batches)
265     k = 1
266     cost = 0
267     for batch in batches:
268         iter = (epoch * B) + k
269         x = batch["x"]
270         y = batch["y"]
271         dropout = self.init_dropout(x.shape[1])
272         cache = self.forward_propagation(params, x, dropout)
273         batch_cost = self.cost_function(params, cache["a"][self.L], y, lambda)
274         cost += x.shape[1] * batch_cost
275         self.backward_propagation(params, cache, y, dropout)
276         self.update_parameters(params, adams, learning_rate, iter)
277         k += 1
278     cost /= data["x"].shape[1]
279     costs.append(cost)
280
281     if (print_cost_iter != 0) and (epoch % print_cost_iter == 0):
282         print(f"Cost_after_epoch_{epoch}:_{cost}")
283
284     return params, costs
285
286 def evaluate(self, params, x):
287     """
288     Parameters:
289     -----

```



```

290         params : Dict[LinearParameters]
291         x : array_like
292
293         Returns:
294         -----
295         y_hat : array_like
296         """
297         cache = self.forward_propagation(params, x)
298         a = cache["a"][self.L]
299         y_hat = (~(a < 0.5)).astype(int)
300         return y_hat
301
302     def accuracy(self, params, data):
303         """
304         Parameters:
305         -----
306         params : Dict[LinearParameters]
307         data : Dict[array_like]
308             data['x'] : array_like
309             data['y'] : array_like
310
311         Returns:
312         -----
313         accuracy : float
314         """
315         x = data["x"]
316         y = data["y"]
317
318         y_hat = self.evaluate(params, x)
319         acc = np.sum(y_hat == y) / y.shape[1]
320
321         return acc

```

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 via numpy

```
1 #! python3
2
3 import numpy as np
4
5 from mllib.utils import LinearParameters, ShuffleBatchData
6 from mllib.utils import apply_activation
7
8
9 def learning_rate_decay_rational(epoch, eta=1.0, alpha=0.2):
10     """
11     Parameters:
12     -----
13     epoch : int
14     eta : float
15         Default = 1.0
16     alpha : float
17         Default = 0.2
18
19     Returns:
20     learning_rate : float
21     """
22     learning_rate = alpha / (1 + eta * epoch)
23     assert (
24         0 <= learning_rate <= 1
25     ), f"learnining_rate_is_outside_[0,1]_for_epoch_{epoch}"
26     return learning_rate
27
28
29 def learning_rate_decay_exponential(epoch, eta=0.95, alpha=0.2):
30     """
31     Parameters:
32     -----
33     epoch : int
34     eta : float
35         Default = 0.95
36     alpha : float
37         Default = 0.2
```

```

38
39 Returns:
40 learning_rate : float
41 """
42 learning_rate = alpha * (eta**epoch)
43 assert (
44     0 <= learning_rate <= 1
45 ), f"learnining_rate_is_outside_[0,1]_for_epoch_{epoch}"
46 return learning_rate
47
48
49 def learning_rate_decay_root(epoch, eta=1.0, alpha=0.2):
50     """
51     Parameters:
52     -----
53     epoch : int
54     eta : float
55         Default = 1.0
56     alpha : float
57         Default = 0.2
58
59     Returns:
60     learning_rate : float
61     """
62     learning_rate = alpha * eta / np.sqrt(epoch + 1)
63     assert (
64         0 <= learning_rate <= 1
65     ), f"learnining_rate_is_outside_[0,1]_for_epoch_{epoch}"
66     return learning_rate
67
68
69 class NeuralNetwork:
70     def __init__(self, config):
71         """
72         Parameters:
73         -----
74         config : Dict
75             config['lp_reg'] = 0,1,2
76             config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
77             config['nodes'] = List[int]
78             config['bias'] = List[Boolean]
79             config['activators'] = List[str]
80             config['keep_probs'] = List[float]
81
82         Returns:
83         -----
84         None

```

```

85         """
86         self.config = config
87         self.lp_reg = config["lp_reg"]
88         self.batch_size = config["batch_size"]
89         self.nodes = config["nodes"]
90         self.bias = config["bias"]
91         self.activators = config["activators"]
92         self.keep_probs = config["keep_probs"]
93         self.L = len(config["nodes"]) - 1
94
95     def init_dropout(self, num_examples, seed=101011):
96         """
97         Parameters:
98         -----
99         num_examples : int
100        seed : int
101            Default: 1 # For reproducibility
102
103        Returns:
104        -----
105        D : Dict[layer : array_like]
106        """
107        np.random.seed(seed)
108        D = {}
109        for l in range(self.L + 1):
110            D[l] = np.random.rand(self.nodes[l], num_examples)
111            D[l] = (D[l] < self.keep_probs[l]).astype(int)
112            D[l] = D[l] / self.keep_probs[l]
113            assert D[l].shape == (
114                self.nodes[l],
115                num_examples,
116            ), "Dropout_matrices_are_the_wrong_shape"
117
118        return D
119
120     def forward_propagation(self, params, x, dropout=None):
121         """
122         Parameters:
123         -----
124         params : Dict[class[Parameters]]
125             params[l].w = Weights
126             params[l].bias = Boolean
127             params[l].b = Bias
128         x : array_like
129
130        Returns:
131        -----

```

```

132         cache = Dict[array_like]
133         cache['a'] = a
134         cache['dg'] = dg
135
136         """
137         # Initialize dictionaries
138         a = {}
139         dg = {}
140
141         a[0], dg[0] = apply_activation(x, self.activators[0])
142         if dropout != None:
143             a[0] = dropout[0] * a[0]
144
145         for l in range(1, self.L + 1):
146             z = params[l].forward(a[l - 1])
147             a[l], dg[l] = apply_activation(z, self.activators[l])
148             if dropout != None:
149                 a[l] = dropout[l] * a[l]
150
151         cache = {"a": a, "dg": dg}
152         return cache
153
154     def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
155         """
156         Parameters:
157         -----
158         params: Dict[LinearParameters]
159         a: array_like
160         y: array_like
161         lambda_: float
162             Default: 0.01
163         eps: float
164             Default: 1e-8
165
166         Returns:
167         -----
168         cost: float
169         """
170         n = y.shape[1]
171         if self.lp_reg == 0:
172             lambda_ = 0.0
173
174         # Compute regularization term
175         R = 0
176         for param in params.values():
177             R += np.sum(np.abs(param.w) ** self.lp_reg)
178         R *= lambda_ / (2 * n)

```

```

179
180     # Compute unregularized cost
181     a = np.clip(a, eps, 1 - eps) # Bound a for stability
182     J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
183
184     cost = float(np.squeeze(J + R))
185
186     return cost
187
188 def backward_propagation(self, params, cache, y, dropout):
189     """
190     Parameters:
191     -----
192     params : Dict[LinearParameters]
193             params[l].w = Weights
194             params[l].bias = Boolean
195             params[l].b = Bias
196     cache : Dict[array_like]
197             cache['a'] : array_like
198             cache['dg'] : array_like
199     y : array_like
200
201     Returns:
202     -----
203     None
204     """
205
206     # Retrieve cache
207     a = cache["a"]
208     dg = cache["dg"]
209
210     # Initialize differentials along the network
211     delta = {}
212     delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
213
214     for l in reversed(range(1, self.L + 1)):
215         delta[l - 1] = (
216             dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
217         )
218
219 def update_parameters(self, params, learning_rate):
220     """
221     Parameters:
222     -----
223     params : Dict[class[Parameters]]
224             params[l].w = Weights
225             params[l].b = Bias

```

```

226         learning_rate : float
227
228     Returns:
229     -----
230     None
231     """
232     for param in params.values():
233         param.update(learning_rate)
234
235     def fit(
236         self,
237         data,
238         eta=1,
239         alpha=0.2,
240         lambda_=0.01,
241         num_epochs=10000,
242         print_cost_iter=1000,
243     ):
244         """
245         Parameters:
246         -----
247         data : Dict[array_like]
248             data['x'] : array_like
249             data['y'] : array_like
250         eta : float
251             Default = 0.1
252         alpha : float
253             Default = 0.1
254         lambda_ : float
255             Default = 0.01
256         num_epochs : int
257             Default = 10000
258         print_cost_iter : int
259             Default = 1000 # 0 Doesn't print costs
260
261     Returns:
262     -----
263     costs : List[floats]
264     params : Dict[LinearParameters]
265     """
266     # Initialize parameters per layer
267     params = {}
268     for l in range(1, self.L + 1):
269         params[l] = LinearParameters(
270             (self.nodes[l], self.nodes[l - 1]), self.bias[l]
271         )
272

```

```

273         # Initialize batching
274         batching = ShuffleBatchData(data, self.batch_size)
275
276         costs = []
277         for epoch in range(num_epochs):
278             batches = batching.get_batches()
279             cost = 0
280             learning_rate = learning_rate_decay_rational(epoch, eta, alpha)
281             for batch in batches:
282                 x = batch["x"]
283                 y = batch["y"]
284                 dropout = self.init_dropout(x.shape[1])
285                 cache = self.forward_propagation(params, x, dropout)
286                 batch_cost = self.cost_function(params, cache["a"][self.L], y, lambda)
287                 cost += x.shape[1] * batch_cost
288                 self.backward_propagation(params, cache, y, dropout)
289                 self.update_parameters(params, learning_rate)
290             cost /= data["x"].shape[1]
291             costs.append(cost)
292
293             if (print_cost_iter != 0) and (epoch % print_cost_iter == 0):
294                 print(f"Cost_after_epoch_{epoch}:_{cost}")
295
296         return params, costs
297
298     def evaluate(self, params, x):
299         """
300         Parameters:
301         -----
302         params : Dict[LinearParameters]
303         x : array_like
304
305         Returns:
306         -----
307         y_hat : array_like
308         """
309         cache = self.forward_propagation(params, x)
310         a = cache["a"][self.L]
311         y_hat = (~(a < 0.5)).astype(int)
312         return y_hat
313
314     def accuracy(self, params, data):
315         """
316         Parameters:
317         -----
318         params : Dict[LinearParameters]
319         data : Dict[array_like]

```



```

320         data['x'] : array_like
321         data['y'] : array_like
322
323     Returns:
324     -----
325     accuracy : float
326     """
327     x = data["x"]
328     y = data["y"]
329
330     y_hat = self.evaluate(params, x)
331     acc = np.sum(y_hat == y) / y.shape[1]
332
333     return acc
334
335
336 if __name__ == "__main__":

```

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.

9 Batch Normalization

See [7].

We recall feature-normalization: Suppose $x \in \mathbb{R}^{n \times N}$ 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 $\gamma, \beta \in \mathbb{R}^n$, 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

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

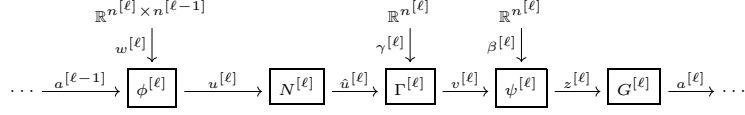
and so the new mean would be given by β . Similarly,

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

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 $\gamma^{[\ell]}, \beta^{[\ell]} \in \mathbb{R}^{n^{[\ell]}}$ and some $\epsilon > 0$, arbitrarily small and prescribed for numerical stability, we define the *batch-normalization* map $BN_{\gamma^{[\ell]}, \beta^{[\ell]}} : \mathbb{R}^{n^{[\ell]}} \rightarrow \mathbb{R}^{n^{[\ell]}}$ 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 $n^{[\ell]}$ nodes, and we focus on the ℓ -th layer specifically to expand:



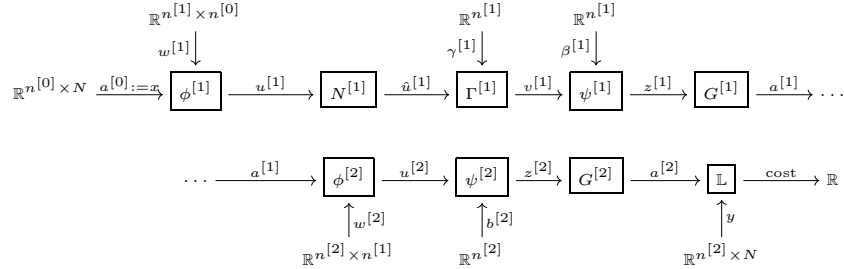
We note that we've dropped the bias term $b^{[\ell]}$ in the above, forward-propagating diagram. If we had included the term, the composition would result in the following

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

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

9.1 Backward Propagation

We consider 2-layer, neural network utilizing batch normalization of the form



where we have the functions

1.

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

is the given loss function. If we're working with a binary classification problem, then we have that

$$\begin{aligned} \mathbb{L}(y, \hat{y}) &= -\frac{1}{N} \sum_{j=1}^n \{y_j \log \hat{y}_j + (1 - y_j) \log(1 - \hat{y}_j)\} \\ &= -\frac{1}{N} [\langle y, \log y \rangle_{\mathbb{R}^N} + \langle 1 - y, \log(1 - \hat{y}) \rangle_{\mathbb{R}^N}]. \end{aligned}$$

2.

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

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

3.

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

is given by

$$\phi^{[\ell]}(w, x) = wx.$$

4.

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

is given by

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

where

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

5.

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

is the normalization operator given by

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

where \mathbb{E} is the expectation operator of a random vector, i.e.,

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

and \mathbb{V} is the variance operator of a random vector, i.e.,

$$\mathbb{V}[x] = \mathbb{E}[(x - \mathbb{E}[x]\vec{1}^T)^{\odot 2}],$$

where $\vec{1} \in \mathbb{R}^N$ and $\odot 2$ represents the Hadamard-square.

6.

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

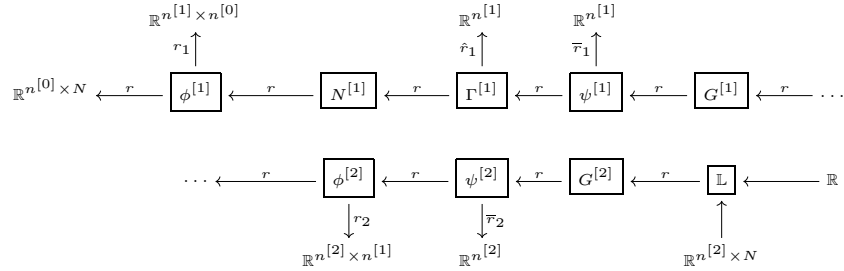
is given by

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

where

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

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



We 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 $\eta \in T_w\mathbb{R}^{m \times n}$, we have that

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

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

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

where

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

Then we look at the two differential paths and for any $(b, x) \in \mathbb{R}^n \times \mathbb{R}^{n \times N}$ any 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

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

with reverse differential

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

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

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

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

(a) In the network direction, we have that

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

with reverse differential

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

(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 appendix, [Section B](#).
6. For the loss function $\mathbb{L} : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}$ given by

$$\mathbb{L}(y, \hat{y}) = -\frac{1}{N} [\langle y, \log \hat{y} \rangle + \langle 1 - y, \log(1 - \hat{y}) \rangle],$$

we fix $y, \hat{y} \in \mathbb{R}^N$ and for $\xi \in T_{\hat{y}}\mathbb{R}^N$, we see that

$$\begin{aligned}d\mathbb{L}_{(y,\hat{y})}(\xi) &= -\frac{1}{N} \sum_{j=1}^N \left[\frac{y_j}{\hat{y}_j} - \frac{1-y_j}{1-\hat{y}_j} \right] \xi_j \\ &= -\frac{1}{N} \left\langle \frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}, \xi \right\rangle,\end{aligned}$$

and hence for $\zeta \in T_{\mathbb{L}(y,\hat{y})}\mathbb{R}$, it follows that

$$r\mathbb{L}_{(y,\hat{y})}(\zeta) = -\frac{1}{N} \left[\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}} \right] \zeta,$$

where the division is taken in the Hadamard sense.

We're now ready to compute our various gradients of our cost function. That is, if we let

$$\mathbb{J} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]} \times n^{[1]}} \times \mathbb{R}^{n^{[1]}} \times \mathbb{R}^{n^{[1]}} \times \mathbb{R}^{n^{[1]} \times n^{[0]}} \rightarrow \mathbb{R}$$

be given by

$$\mathbb{J}(p) = \mathbb{L}(y, G^{[2]} \circ \psi^{[2]}(b^{[2]}, \phi^{[2]}(w^{[2]}, G^{[2]} \circ \psi^{[2]}(\beta^{[1]}, \Gamma^{[1]}(\gamma^{[1]}, N^{[1]} \circ \phi^{[1]}(W^{[1]}, x))))),$$

where $p = (w^{[2]}, \gamma^{[1]}, \beta^{[1]}, w^{[2]}, b^{[2]})$ is a point in our parameter-space and we compute the reverse differentials for a learning rate $\alpha \in T_{\mathbb{J}}\mathbb{R}$ with the assumption that our second activator function is the sigmoid function. Indeed,

$$\begin{aligned}
r(\mathbb{L} \circ G^{[2]})_v(\alpha) &= rG_v^{[2]} \circ r\mathbb{L}_a(\alpha) \\
&= -\frac{\alpha}{N} G^{[2]'}(v) \odot \left[\frac{y}{a} - \frac{1-y}{1-a} \right] \\
&= -\frac{\alpha}{N} a(1-a) \left[\frac{y}{a} - \frac{1-y}{1-a} \right] \\
&= -\frac{\alpha}{N} [y(1-a) - a(1-y)] \\
&= -\frac{\alpha}{N} [y - a] \\
&= \frac{a-y}{N} \alpha.
\end{aligned}$$

This leads us to

$$\begin{aligned}
\bar{r}_2 \mathbb{J}_{b^{[2]}}(\alpha) &= \bar{r}_2(\psi^{[2]})_{(b^{[2]}, u^{[2]})} \circ rG_{v^{[2]}}^{[2]} \circ r\mathbb{L}_{(y, a^{[2]})} \\
&= \frac{\alpha}{N} R_{\bar{1}}(a^{[2]} - y) \\
&= \frac{\alpha}{N} \sum_{j=1}^N (a^{[2]}_j - y_j);
\end{aligned}$$

$$\begin{aligned}
r_2 \mathbb{J}_{W^{[2]}}(\alpha) &= r_2 \phi_{(W^{[2]}, a^{[1]})}^{[2]} \circ r\psi_{(b^{[2]}, u^{[2]})}^{[2]} \left(\frac{\alpha}{N} (a^{[2]} - y) \right) \\
&= r_2 \phi_{(W^{[2]}, a^{[1]})}^{[2]} \left(\frac{\alpha}{N} (a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} (a^{[2]} - y) a^{[1]T};
\end{aligned}$$

$$\begin{aligned}
\bar{r}_1 \mathbb{J}_{\beta^{[1]}}(\alpha) &= \bar{r}_1 \psi_{(\beta^{[1]}, \hat{z}^{[1]})}^{[1]} \circ rG_{\hat{z}^{[1]}}^{[1]} \circ r\phi_{(W^{[2]}, a^{[2]})}^{[2]} \circ r\psi_{(b^{[2]}, u^{[2]})}^{[2]} \circ r(\mathbb{L} \circ G^{[2]})_{v^{[2]}}(\alpha) \\
&= \frac{\alpha}{N} \bar{r}_1 \psi_{(\beta^{[1]}, \hat{z}^{[1]})}^{[1]} \circ rG_{\hat{z}^{[1]}}^{[1]} \circ r\phi_{(W^{[2]}, a^{[2]})}^{[2]} (a^{[2]} - y) \\
&= \frac{\alpha}{N} \bar{r}_1 \psi_{(\beta^{[1]}, \hat{z}^{[1]})}^{[1]} \circ rG_{\hat{z}^{[1]}}^{[1]} (W^{[2]T} (a^{[2]} - y)) \\
&= \frac{\alpha}{N} \bar{r}_1 \psi_{(\beta^{[1]}, \hat{z}^{[1]})}^{[1]} (G^{[1]'}(\hat{z}^{[1]}) \odot W^{[2]T} (a^{[2]} - y)) \\
&= \frac{\alpha}{N} \sum_{j=1}^N g^{[1]'}(\hat{z}^{[1]}_j) \odot W^{[2]T} (a^{[2]}_j - y_j);
\end{aligned}$$

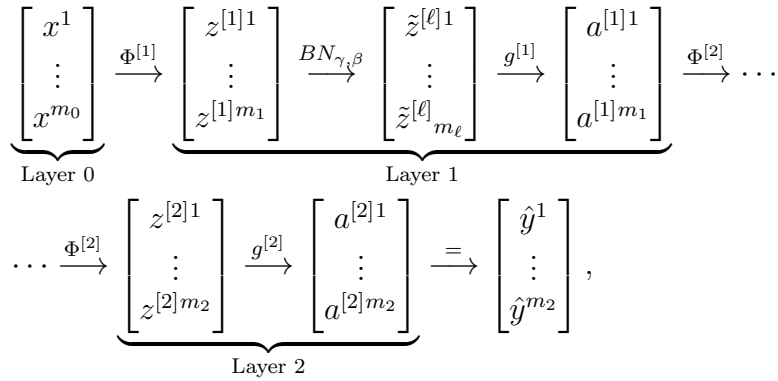
$$\begin{aligned}
\hat{r}_1 \mathbb{J}_{\gamma^{[1]}}(\alpha) &= \frac{\alpha}{N} \hat{r}_1 \Gamma_{(\gamma^{[1]}, z^{[1]})}^{[1]} (G^{[1]'}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)) \\
&= \frac{\alpha}{N} R_{\tilde{\Gamma}}(z^{[1]} \odot (G^{[1]'}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y))) \\
&= \frac{\alpha}{N} \sum_{j=1}^n z^{[1]}_j \odot g^{[1]'}(\tilde{z}^{[1]}_j) \odot W^{[2]T}(a^{[2]}_j - y_j);
\end{aligned}$$

and finally,

$$\begin{aligned}
r_1 \mathbb{J}_{W^{[1]}}(\alpha) &= \frac{\alpha}{N} r_1 \phi_{(W^{[1]}, x)}^{[1]} \circ r N_{u^{[1]}}^{[1]} \circ r \Gamma_{(\gamma^{[1]}, z^{[1]})}^{[1]} (G^{[1]'}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y)) \\
&= \frac{\alpha}{N} r_1 \phi_{(W^{[1]}, x)}^{[1]} \circ r N_{u^{[1]}}^{[1]} \left(\gamma \tilde{\Gamma}^T \odot G^{[1]'}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} R_{x^T} \circ r N_{u^{[1]}}^{[1]} \left(\gamma \tilde{\Gamma}^T \odot G^{[1]'}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} \sum_{j,l=1}^N \sum_{i=1}^{n^{[1]}} T_i^{jk} \gamma^i g^{[1]'}(\tilde{z}^{[1]}_j) W^{[2]}_{i,l} (a^{[2]}_j - y_j) x_l^m
\end{aligned}$$

9.2 Backward Propagation - OLD

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 ??



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.

9.3 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.

9.4 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

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

9.5 Python Implementation

Work in Progress

Appendices

A The Reverse Differential

In order to apply gradient descent to our trainable parameters, we obviously have a need to compute various gradients of the cost function which is essentially a large functional composition. Computing intermediate gradients along this computation doesn't make sense mathematically as stated. However, the usual exterior derivative works very well in this context. However, since we would like to vectorize this process, the exterior derivative falls short for our implementation purposes. This leads us to a related form of differentiation, namely, the reverse derivative. We give here a brief exposition of the reverse differential in the setting of Riemannian geometry, and then use Euclidean spaces as our examples. C.f., [1], [2], [3], [4], [5], [6], [8], [9], [10], [11], [12].

We first recall the definition of the exterior derivative between smooth manifolds.

Definition A.1. *Suppose M, N are smooth manifolds and $f : M \rightarrow N$ is smooth. Then for $p \in M$, the (exterior) differential of f at p , denoted df_p , is the linear map*

$$df_p : T_p M \rightarrow T_{f(p)} N$$

, such that for any $\xi \in T_p M$ and any $g \in C^\infty(N)$, we have that

$$df_p(\xi)[g] = \xi[g \circ f].$$

Example A.2. *Suppose $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is smooth with coordinates (x^j) on \mathbb{R}^n and coordinates (y^j) on \mathbb{R}^m . Then at a point $p \in \mathbb{R}^n$, we have the differential in coordinates*

$$df_p = \frac{y^i \circ f}{\partial x^j}(p) dx^j \Big|_p \otimes \frac{\partial}{\partial y^i} \Big|_{f(p)}.$$

In matrix form, we have the Jacobian representation of df_p , denoted $Jf_p \in \mathbb{R}^{m \times n}$, given by

$$Jf_p = \begin{bmatrix} \frac{\partial f^1}{\partial x^1} \Big|_p & \cdots & \frac{\partial f^1}{\partial x^n} \Big|_p \\ \frac{\partial f^2}{\partial x^1} \Big|_p & \cdots & \frac{\partial f^2}{\partial x^n} \Big|_p \\ \vdots & \ddots & \vdots \\ \frac{\partial f^m}{\partial x^1} \Big|_p & \cdots & \frac{\partial f^m}{\partial x^n} \Big|_p \end{bmatrix},$$

where $f^i := y^i \circ f$.

Moreover, for any fixed $p \in \mathbb{R}^n$, we may identify \mathbb{R}^n with the tangent space $T_p\mathbb{R}^n$ via

$$v = (v^1, \dots, v^n) \in \mathbb{R}^n \rightsquigarrow \vec{v} = v^j \frac{\partial}{\partial x^j} \Big|_p \in T_p\mathbb{R}^n.$$

It then follows that

$$\begin{aligned} df_p(\vec{v}) &= v^j \frac{\partial f^i}{\partial x^j} \Big|_p \frac{\partial}{\partial y^i} \Big|_{f(p)} \\ &\rightsquigarrow \left(v^j \frac{\partial f^1}{\partial x^j} \Big|_p, \dots, v^j \frac{\partial f^m}{\partial x^j} \Big|_p \right) \\ &= Jf_p v \end{aligned}$$

reverseDifferential

Definition A.3. Suppose (M, g) and (N, h) are Riemannian manifolds and suppose $f : M \rightarrow N$ is smooth. Then for $p \in M$, the reverse differential, denoted rf_p , is the linear map

$$rf_p : T_{f(p)}M \rightarrow T_pM$$

such that for any $\xi \in T_pM$ and any $\zeta \in T_{f(p)}N$, the following equality holds

$$g(rf_p(\zeta), \xi) = h(\zeta, df_p(\xi)).$$

Example A.4. Suppose $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is smooth with coordinates (x^j) on \mathbb{R}^n and coordinates (y^j) on \mathbb{R}^m . Then at a point $p \in \mathbb{R}^n$, we have the reverse differential in coordinates

$$rf_p = \sum_{i=1}^m \sum_{j=1}^n \frac{\partial f^i}{\partial x^j} \Big|_p dy^i \Big|_{f(p)} \otimes \frac{\partial}{\partial x^j} \Big|_p,$$

where $f^i := y^i \circ f$.

In matrix form, we have the Jacobian representation of rf_p , denoted $J^T f_p \in \mathbb{R}^{n \times m}$, given by

$$J^T f_p = \begin{bmatrix} \frac{\partial f^1}{\partial x^1} \Big|_p & \cdots & \frac{\partial f^m}{\partial x^1} \Big|_p \\ \frac{\partial f^1}{\partial x^2} \Big|_p & \cdots & \frac{\partial f^m}{\partial x^2} \Big|_p \\ \vdots & \ddots & \vdots \\ \frac{\partial f^1}{\partial x^n} \Big|_p & \cdots & \frac{\partial f^m}{\partial x^n} \Big|_p \end{bmatrix}$$

Moreover, for $w \in \mathbb{R}^m \rightsquigarrow \vec{w} \in T_{f(p)}\mathbb{R}^m$ and $v \in \mathbb{R}^n \rightsquigarrow \vec{v} \in T_p\mathbb{R}^n$, it follows that

$$\begin{aligned} \langle rf_p(\vec{w}), \vec{v} \rangle_{T_p\mathbb{R}^n} &= \langle \vec{w}, df_p(\vec{v}) \rangle_{T_{f(p)}\mathbb{R}^m} \\ &= \langle w, Jf_p(v) \rangle_{\mathbb{R}^m} \\ &= \langle J^T f_p(w), v \rangle_{\mathbb{R}^n}, \end{aligned}$$

and hence that

$$rf_p(\vec{w}) = J^T f_p(w).$$

Proposition A.5. Suppose we have the compositional diagram

$$(M, g) \xrightarrow{\phi} (N, h) \xrightarrow{\psi} (Q, k)$$

and we let $f := \psi \circ \phi : (M, g) \rightarrow (Q, k)$. Then for any $p \in M$, the reverse derivative satisfies

$$rf_p = r\phi_p \circ r\psi_{\phi(p)}.$$

Proof: Fix $p \in M$, and let $\xi \in T_p M$ and $\zeta \in T_{f(p)} Q$. Then we have that

$$\begin{aligned} g(rf_p(\zeta), \xi) &= k(\zeta, df_p(\xi)) \\ &= k(\zeta, d\psi_{\phi(p)} \circ d\phi_p(\xi)) \\ &= h(r\psi_{\phi(p)}(\zeta), d\phi_p(\xi)) \\ &= g(r\phi_p(r\psi_{\phi(p)}(\zeta)), \xi) \\ &= g(r\phi_p \circ r\psi_{\phi(p)}(\zeta), \xi), \end{aligned}$$

as desired. □

The following needs to be refined further still.

Example A.6. Suppose $f : (\mathbb{R}^{m \times n}, (X_j^i), F) \rightarrow (\mathbb{R}, (t), \delta)$ is smooth, where F is the Frobenius inner product. Suppose $v \in T_P \mathbb{R}^{m \times n} \rightsquigarrow V \in \mathbb{R}^{m \times n}$ are represented via

$$v = v_j^i \left. \frac{\partial}{\partial X_j^i} \right|_P \rightsquigarrow V = [v_j^i],$$

and in coordinates, we have that

$$df_P = \left. \frac{\partial f}{\partial X_j^i} \right|_P dX_j^i|_P.$$

The matrix-Jacobian-representation of f at P , denoted $Jf_P \in \mathbb{R}^{m \times n}$ is given by

$$Jf_P = \begin{bmatrix} \left. \frac{\partial f}{\partial X_1^1} \right|_P & \cdots & \left. \frac{\partial f}{\partial X_n^1} \right|_P \\ \left. \frac{\partial f}{\partial X_1^2} \right|_P & \cdots & \left. \frac{\partial f}{\partial X_n^2} \right|_P \\ \vdots & \ddots & \vdots \\ \left. \frac{\partial f}{\partial X_1^m} \right|_P & \cdots & \left. \frac{\partial f}{\partial X_n^m} \right|_P \end{bmatrix}.$$

It then follows that

$$\begin{aligned} df_P(v) &= v_j^i \left. \frac{\partial f}{\partial X_j^i} \right|_P \\ &= \langle V, Jf_P \rangle_{F(m,n)}. \end{aligned}$$

Similarly, if $\tau \in \mathbb{R} \longleftrightarrow \vec{\tau} = \tau \frac{d}{dt} \Big|_{f(P)} \in T_{f(P)}\mathbb{R}$, we see the reverse differential given in coordinates

$$rf_P = \sum_{i=1}^m \sum_{j=1}^n \left. \frac{\partial f}{\partial X_j^i} \right|_P dt|_P \otimes \left. \frac{\partial}{\partial X_j^i} \right|_{f(P)},$$

evaluates to

$$rf_P(\vec{\tau}) = \tau \sum_{i=1}^m \sum_{j=1}^n \left. \frac{\partial f}{\partial X_j^i} \right|_P \left. \frac{\partial}{\partial X_j^i} \right|_{f(P)},$$

and hence that

$$\begin{aligned} \langle rf_P(\vec{\tau}), v \rangle_{T_P \mathbb{R}^{m \times n}} &= \langle \vec{\tau}, df_P(v) \rangle_{T_{f(P)} \mathbb{R}} \\ &= \tau df_P(v) \\ &= \tau \langle V, Jf_P \rangle_{F(m,n)} \end{aligned}$$

Lemma A.7. 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 $R \in \mathbb{R}_n^k{}^m$ 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. □

B The Normalization Operator

sec:normOp

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

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

B.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)

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

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] Henk P Barendregt and Erik Barendsen. Introduction to lambda calculus. In *Aspens Workshop on Implementation of Functional Languages, Göteborg. Programming Methodology Group, University of Göteborg and Chalmers University of Technology*, volume 85, 1988.
- [2] Richard F Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential categories. *Theory and Applications of Categories*, 22(23):622–672, 2009.
- [3] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories. *arXiv preprint arXiv:1910.07065*, 2019.
- [4] Geoffrey SH Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In *European Symposium on Programming*, pages 1–28. Springer, Cham, 2022.
- [5] Brendan Fong, David Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.
- [6] Bruno Gavranović. Compositional deep learning. *arXiv preprint arXiv:1907.08292*, 2019.
- [7] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.
- [8] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.
- [9] Carol Mak and Luke Ong. A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. *arXiv preprint arXiv:2002.08241*, 2020.
- [10] Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010.

- [11] Dan Shieber, Bruno Gavranović, and Paul Wilson. Category theory in machine learning. *arXiv preprint arXiv:2106.07032*, 2021.
- [12] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.