Neural Networks

Matt R

August 1, 2022

Contents

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 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 $\sigma : \mathbb{R} \to (0,1)$ 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

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

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

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

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

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

1.1 The Gradient

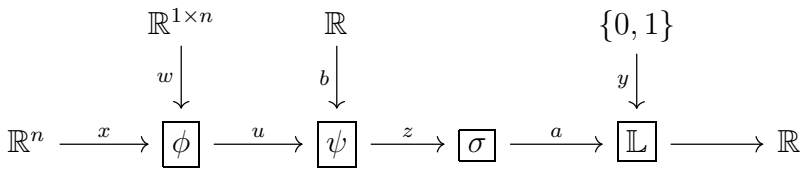
We wish to compute the gradient of our cost function $\mathbb{J}$ with respect to our trainable parameters, $w \in \mathbb{R}^{1\times n}$ and $b \in \mathbb{R}$. To this end, we define the functions

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

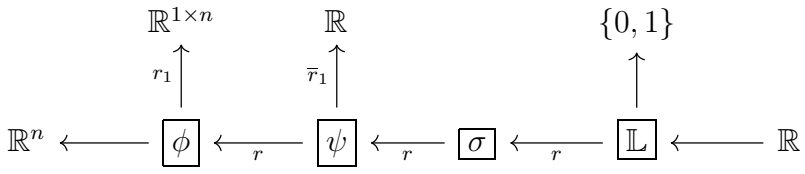
and

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

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.
$$\phi : \mathbb{R}^{1\times n} \times \mathbb{R}^n \to \mathbb{R}, \qquad u := \phi(w, x) = wx.$$
Then for 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
$$\begin{aligned} \left\langle r_1\phi_{(w,x)}(\zeta), \eta \right\rangle_{\mathbb{R}^{1\times n}} &= \left\langle \zeta, d_1\phi_{(w,x)}(\eta) \right\rangle_{\mathbb{R}} \\ &= \left\langle \zeta, R_x(\eta) \right\rangle_{\mathbb{R}} \\ &= \left\langle R_{x^T}(\zeta), \eta \right\rangle_{\mathbb{R}^{1\times n}}, \end{aligned}$$
and hence that
$$r_1\phi_{(w,x)} = R_{x^T}.$$

2.
$$\psi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \qquad z := \psi(b, u) = u + b.$$
Then for any $(b, u) \in \mathbb{R} \times \mathbb{R}$ and any $\xi \in T_u\mathbb{R}$, we have that
$$d\psi_{(b,u)}(\xi) = \mathbb{1}_{\mathbb{R}}(\xi),$$
and similarly for any $\eta \in T_b\mathbb{R}$, we have that
$$\overline{d}_1\psi_{(b,u)}(\eta) = \mathbb{1}_{\mathbb{R}}(\eta).$$
We then immediately have that
$$r\psi_{(b,u)} = \mathbb{1}_{\mathbb{R}},$$
and
$$\overline{r}_1\psi_{(b,u)} = \mathbb{1}_{\mathbb{R}}.$$

3.
$$\sigma : \mathbb{R} \to \mathbb{R}, \qquad a := \sigma(z) = \frac{1}{1 + e^{-z}}.$$

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.

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

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

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

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.

```
#! python3

import numpy as np

from mlLib.utils import apply_activation


class LinearParameters:
    def __init__(self, dims, bias=True, seed=1):
        """
        Parameters:
        -----------
        dims : tuple(int, int)
        bias : Boolean
            Default : True
        seed : int
            Default : 1

        Returns:
        --------
        None
        """
        np.random.seed(seed)
        self.dims = dims
        self.bias = bias
        self.w = np.random.randn(*dims) * 0.01
        if bias:
            self.b = np.zeros((dims[0], 1))

    def forward(self, x):
        """
        Parameters:
        -----------
        x : array_like

        Returns:
        --------
        z : array_like
        """
        z = np.einsum("ij,jk", self.w, x)
        if self.bias:
            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`

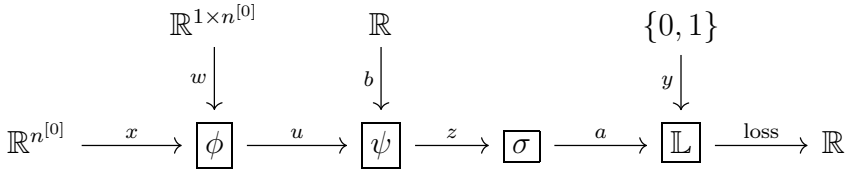
Here we include the general method of coding a logistic regression model via `scikit-learn`'s modeling library.

```
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
```

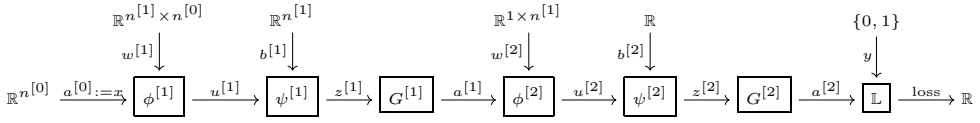
```
    x = dataset[:, :10]
    y = dataset[:, 10]

    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
    mu = np.mean(x, axis=0, keepdims=True)
    var = np.var(x, axis=0, keepdims=True)
    x_train = (x_train - mu) / np.sqrt(var)
    x_test = (x_test - mu) / np.sqrt(var)

    log_reg = LogisticRegression()
    log_reg.fit(x_train, y_train)
    train_acc = log_reg.score(x_train, y_train)
    print(f'The accuracy on the training set: {train_acc}.')
    test_acc = log_reg.score(x_test, y_test)
    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)

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

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

Definition 2.1. *Suppose $g : \mathbb{R} \to \mathbb{R}$ is any function. Then we say $G : \mathbb{R}^n \to \mathbb{R}^n$ 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 $g : \mathbb{R} \to \mathbb{R}$ is any smooth function and $G : \mathbb{R}^n \to \mathbb{R}^n$ is the broadcasting of g from $\mathbb{R}$ to $\mathbb{R}^n$. Then the differential $dG_z : T_z\mathbb{R}^n \to T_{G(z)}\mathbb{R}^n$ 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

$$[dG_z]^i_j = \delta^i_j g'(z^i).$$

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 $e_1, ...e_m$ 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. □

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

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

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

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

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 $\mathrm{ReLU}(z; \beta)$ given by

$$\mathrm{ReLU} : \mathbb{R} \to \mathbb{R}, \qquad \mathrm{ReLU}(z; \beta) = \max\{\beta z, z\},$$

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.,

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

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

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

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 $\mathrm{softmax}(z)$, and so we calculate

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

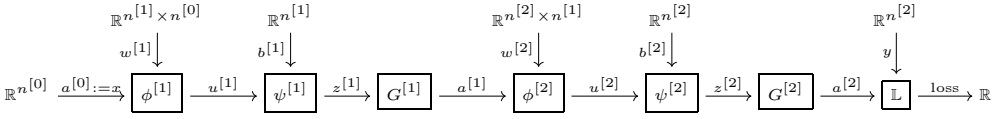
That is, as a map $dS_z : T_z\mathbb{R}^m \to T_{S(z)}\mathbb{R}^m$, we have that

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

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

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

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

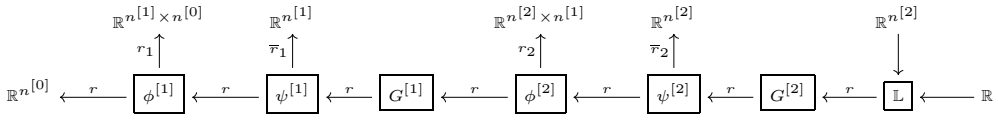
   is given by

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

3.
$$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}} \to \mathbb{R}^{n^{[\ell]}}$$
is given by
$$\psi^{[\ell]}(b, x) = x + b.$$

4.
$$\mathbb{L} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]}} \to \mathbb{R}$$
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 $G : \mathbb{R}^n \to \mathbb{R}^n$ is the broadcasting of $g : \mathbb{R} \to \mathbb{R}$. Then for $(x, \xi) \in T\mathbb{R}^n$, we have that
$$\begin{aligned} dG_x(\xi) &= G'(x) \odot \xi \\ &= \operatorname{diag}(G'(x)) \cdot \xi \end{aligned}$$
and for any $\zeta \in T_{G(x)}\mathbb{R}^n$, the reverse derivative is given by
$$\begin{aligned} rG_x(\zeta) &= G'(x) \odot \zeta \\ &= \operatorname{diag}(G'(x)) \cdot \zeta. \end{aligned}$$

2. Suppose $\phi : \mathbb{R}^{m \times n} \times \mathbb{R}^n \to \mathbb{R}^m$ 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 $\psi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ 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

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

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

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

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

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

and hence

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

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

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

and hence

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

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

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

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

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

and hence

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

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

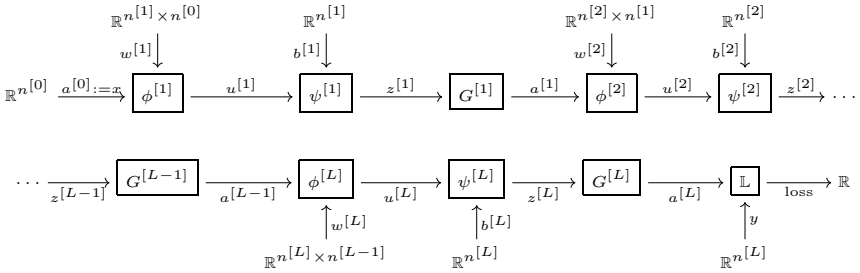
$$\begin{aligned} r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1\phi^{[1]}_p \circ r\psi^{[1]}_{u^{[1]}} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{a^{[1]}} \circ r\psi^{[2]}_{u^{[2]}} \circ rG^{[2]}_{z^{[2]}} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{x^T} \circ \mathbb{1} \circ rG^{[1]}_{z^{[1]}} \circ L_{w^{[2]T}} \circ \mathbb{1} \circ rG^{[2]}_{z^{[2]}} \circ r\mathbb{L}_{a^{[2]}} \\ &= R_{x^T} \circ rG^{[1]}_{z^{[1]}} \circ L_{w^{[2]T}} \circ rG^{[2]}_{z^{[2]}} \circ r\mathbb{L}_{a^{[2]}}, \end{aligned}$$

and hence

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

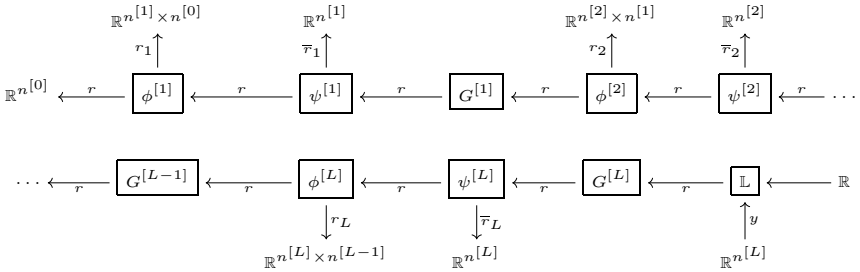
3 Deep Neural Networks

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

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

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

$$\begin{aligned}{\delta^{[L-1]}}_j &:= r(\mathbb{L} \circ G^{[L]} \circ \psi^{[L]} \circ \phi^{[L]} \circ G^{[L-1]})_{{z^{[L-1]}}_j} \\ &= rG^{[L-1]}_{{z^{[L-1]}}_j} \circ r\phi^{[L]}_{(w^{[L]}, {a^{[L-1]}}_j)} \circ r\psi^{[L]}_{(b^{[L]}, {u^{[L]}}_j)} \circ rG^{[L]}_{z_j^{[L]}} \circ r\mathbb{L}_{({a^{[L]}}_j, y_j)} \\ &= G^{[L-1]\prime}({z^{[L-1]}}_j) \odot w^{[L]T} \cdot {\delta^{[L]}}_j.\end{aligned}$$

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

$${\delta^{[\ell]}}_j := G^{[\ell]\prime}({z^{[\ell]}}_j) \odot w^{[\ell+1]T} {\delta^{[\ell+1]}}_j.$$

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.

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation


class NeuralNetwork:
    def __init__(self, config):
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['nodes'] = List[int]
            config['bias'] = List[Boolean]
            config['activators'] = List[str]

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config["lp_reg"]
        self.nodes = config["nodes"]
        self.bias = config["bias"]
        self.activators = config["activators"]
        self.L = len(config["nodes"]) - 1

    def forward_propagation(self, params, x):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
```

```
        --------
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])

        for l in range(1, self.L + 1):
            z = params[l].forward(a[l - 1])
            a[l], dg[l] = apply_activation(z, self.activators[l])

        cache = {"a": a, "dg": dg}
        return cache

    def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params: class[Parameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            Default: 1e-8

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
        if self.lp_reg == 0:
            lambda_ = 0.0

        # Compute regularization term
        R = 0
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        R *= lambda_ / (2 * n)

        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)  # Bound a for stability
```

```
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, params, cache, y):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache["a"]
        dg = cache["dg"]

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = (a[self.L] - y) / y.shape[1]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1])

    def update_parameters(self, params, learning_rate):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].b = Bias
        learning_rate : float

        Returns:
        --------
        None
```

```
        """
        for param in params.values():
            param.update(learning_rate)

    def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
        """
        Parameters:
        -----------
        x : array_like
        y : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

        Returns:
        --------
        costs : List[floats]
        params : class[Parameters]
        """
        # Initialize parameters per layer
        params = {}
        for l in range(1, self.L + 1):
            params[l] = LinearParameters(
                (self.nodes[l], self.nodes[l - 1]), self.bias[l]
            )

        costs = []
        for i in range(num_iters):
            cache = self.forward_propagation(params, x)
            cost = self.cost_function(params, cache["a"][self.L], y, lambda_)
            costs.append(cost)
            self.backward_propagation(params, cache, y)
            self.update_parameters(params, learning_rate)

            if i % 1000 == 0:
                print(f"Cost after iteration {i}: {cost}")

        return params

    def evaluate(self, params, x):
        """
        Parameters:
        -----------
        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)
```

```
    train = {'x' : x_train, 'y' : y_train}
    test = {'x' : x_test, 'y' : y_test}
    mu = np.mean(train['x'], axis=0, keepdims=True)
    var = np.var(train['x'], axis=0, keepdims=True)
    train['x'] = (train['x'] - mu) / np.sqrt(var)
    test['x'] = (test['x'] - mu) / np.sqrt(var)

    ## Define network structure
    input_layer = Input(shape=(10,))
    hidden_layer_1 = Dense(
        32,
        activation='relu',
        kernel_initializer='he_normal',
        bias_initializer='zeros'
    )(input_layer)
    hidden_layer_2 = Dense(
        8,
        activation='relu',
        kernel_initializer='he_normal',
        bias_initializer='zeros'
    )(hidden_layer_1)
    output_layer = Dense(
        1,
        activation='sigmoid',
        kernel_initializer='he_normal',
        bias_initializer='zeros'
    )(hidden_layer_2)

    model = Model(inputs=input_layer, outputs=output_layer)
    model.summary()

    ## Compile desired model
    model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )

    ## Train the model
    hist = model.fit(
        train['x'],
        train['y'],
        batch_size=32,
        epochs=150,
        validation_split=0.17
    )

```

```
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

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

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 $\mathcal{E} : 2^{\mathbb{D}} \to [0, 1]$ by

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

where $\varepsilon : \mathbb{D} \to \{0, 1\}$ 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 $\mathcal{E}(\mathbb{X}) < \mathcal{E}(\mathcal{D}) \lessapprox 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.
- If $\mathcal{E}(\mathbb{X}) \approx \mathcal{E}(\mathcal{D}) \gtrapprox 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.
- If $10\epsilon \lessapprox \mathcal{E}(\mathbb{X}) \ll \mathcal{E}(\mathcal{D})$, 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:

```
import mlLib.npActivators as npActivators

## Classes
## Timing Epoch
class EpochRuntime():
    def __init__(self):
        self.current = time.time()

    def elapsed_time(self):
        elapsed = time.time() - self.current
        mins, secs = elapsed // 60, elapsed % 60
        txt = 'Elapsed time for the most recent epoch: {0} minutes and {1:0.3f}secon
        print(txt)
        self.current = time.time()



## Shuffle, split and normalize full dataset
class ProcessData():
    def __init__(self, x, y, test_percent, dev_percent=0.0, seed=101, shuffle=True,
        """
        Parameters:
        -----------
        x : array_like
            x.shape = (examples, features)
        y : array_like
            y.shape = (examples, labels)
        test_percent : float
        dev_pervents : Tuple(floats)
        seed : int
            Default = 1
        shuffle : Boolean
            Default = True
        feat_as_col : Boolean
            Default = True

        Returns:
        --------
        None
        """
        self.x = x
        self.y = y
        self.test_percent = test_percent
        self.dev_percent = dev_percent
```

```
45          self.seed = seed
46          self.shuffle = shuffle
47          self.feat_as_col = feat_as_col
48
49          self.split()
50          self.normalize()
51
52          print(f"x_train.shape: {self.train['x'].shape}")
53          print(f"y_train.shape: {self.train['y'].shape}")
54          print(f"x_test.shape: {self.test['x'].shape}")
55          print(f"y_test.shape: {self.test['y'].shape}")
56          if self.dev_percent > 0.0:
57              print(f"x_dev.shape: {self.dev['x'].shape}")
58              print(f"y_dev.shape: {self.dev['y'].shape}")
59
60      def split(self):
61          """
62          Parameters:
63          -----------
64          None
65
66          Returns:
67          --------
68          None
69          """
70          x_aux, x_test, y_aux, y_test = train_test_split(self.x, self.y, test_size=se
71          left_over = 1 - self.test_percent
72          aux_perc = self.dev_percent / left_over
73          x_train, x_dev, y_train, y_dev = train_test_split(x_aux, y_aux, test_size=au
74
75          if self.feat_as_col:
76              self.train = {'x' : x_train, 'y' : y_train}
77              self.test = {'x' : x_test, 'y' : y_test}
78              self.dev = {'x' : x_dev, 'y' : y_dev}
79          else:
80              self.train = {'x' : x_train.T, 'y' : y_train.T}
81              self.test = {'x' : x_test.T, 'y' : y_test.T}
82              self.dev = {'x' : x_dev.T, 'y' : y_dev.T}
83
84      def normalize(self, z=None, eps=0.0):
85          """
86          Parameters:
87          -----------
88          z : array_like
89              Default : None - For initialization
90          eps : float
91              Default 0.0 - For stability
```

```
92
93          Returns:
94          z_scale : array_like
95          """
96          if z == None:
97              x = self.train['x']
98              axis = 0 if self.feat_as_col else 1
99              self.mu = np.mean(x, axis=axis, keepdims=True)
100             self.var = np.var(x, axis=axis, keepdims=True)
101             self.theta = 1 / np.sqrt(self.var + eps)
```

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 $\mathbb{L} : \mathbb{R}^{n^{[L]}} \times \mathbb{R}^{n^{[L]}} \to \mathbb{R}$, 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

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

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

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

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

$$\min_{w,b} \mathbb{J}^R(w, b) = \min_{w,b} \left\{ \mathbb{J}(w, b) + R(w) \right\},$$

and so for suitably chosen $\lambda > 0$, it forces $\left\| w^{[\ell]} \right\|_F$ 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

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

which is dependent on λ.

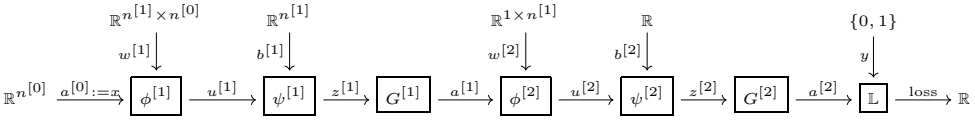
iv. Then using the aforementioned $(w, b) = (w, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.

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 $\mathbb{P}_\ell$ on Q_ℓ by

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

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.

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation


class NeuralNetwork():
    def __init__(self, config):
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['nodes'] = List[int]
```

```
            config['bias'] = List[Boolean]
            config['activators'] = List[str]
            config['keep_probs'] = List[float]

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config['lp_reg']
        self.nodes = config['nodes']
        self.bias = config['bias']
        self.activators = config['activators']
        self.keep_probs = config['keep_probs']
        self.L = len(config['nodes']) - 1

    def init_dropout(self, num_examples, seed=1):
        """
        Parameters:
        -----------
        num_examples : int
        seed : int
            Default: 1 # For reproducability

        Returns:
        --------
        D : Dict[layer : array_like]
        """
        np.random.seed(seed)
        D = {}
        for l in range(self.L + 1):
            D[l] = np.random.rand(self.nodes[l], num_examples)
            D[l] = (D[l] < self.keep_probs[l]).astype(int)
            D[l] = D[l] / self.keep_probs[l]
            assert (D[l].shape == (self.nodes[l], num_examples)), "Dropout matrices

        return D

    def forward_propagation(self, params, x, dropout=None):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        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
```

```
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        R *= (lambda_ / (2 * n))

        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)    # Bound a for stability
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, params, cache, y, dropout):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache['a']
        dg = cache['dg']

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * drop

    def update_parameters(self, params, learning_rate=0.1):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
```

```
            params[l].bias = Boolean
            params[l].b = Bias
        learning_rate : float
            Default : 0.01

        Returns:
        --------
        None
        """
        for param in params.values():
            param.update(learning_rate)

    def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
        """
        Parameters:
        -----------
        x : array_like
        y : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

        Returns:
        --------
        costs : List[floats]
        params : class[Parameters]
        """
        # Initialize parameters per layer
        params = {}
        for l in range(1, self.L + 1):
            params[l] = LinearParameters(
                (self.nodes[l], self.nodes[l - 1]), self.bias[l])

        costs = []
        for i in range(num_iters):
            dropout = self.init_dropout(x.shape[1])
            cache = self.forward_propagation(params, x, dropout)
            cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
            costs.append(cost)
            self.backward_propagation(params, cache, y, dropout)
            self.update_parameters(params, learning_rate)

            if i % 1000 == 0:
                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 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 $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function. Then, we recall the definition of the partial derivative

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

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 $F : \mathbb{R}^n \times (0, 1) \to \mathbb{R}^n$ 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:

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

```
#! python3

import numpy as np
from numpy.linalg import norm

## Checking the reverse differential of a function
def differential_check(f, x, eps=1e-3):
    """
    Parameters:
    -----------
    f : function
    x : array_like
    eps : float
        Default = 10^{-3}

    Returns:
    --------
    error
    """
    y, rf = f(x)
    x = np.array(x)
    if len(x.shape) == 0:
        x = x.reshape(1, 1)
    elif len(x.shape) == 1:
        x = x.reshape(-1, 1)
    if len(y.shape) == 0:
        y = y.reshape(1, 1)
    elif len(y.shape) == 1:
```

```
        y = y.reshape(-1, 1)

    # k, l = y.shape
    m, n = x.shape
    # F = np.zeros((m, n, k, l))
    F = np.zeros((*x.shape, *y.shape))
    rf = rf.reshape(*x.shape, *y.shape)

    for i in range(m):
        for j in range(n):
            e = np.zeros((m, n))
            e[i, j] = 1
            x_plus = x + eps * e
            x_minus = x - eps * e
            f_plus, _ = f(x_plus)
            f_minus, _ = f(x_minus)
            f_diff = f_plus - f_minus
            f_diff = f_diff.reshape(*y.shape)
            F[i, j] = f_diff

    F = F / (2 * eps)

    error = norm(F - rf) / (norm(F) + norm(rf))

    return error


def sigmoid(x):
    ## sigmoid: âĎİ^n âĘŠ âĎİ^n ##
    # n = 1 is valid
    x = np.array(x)
    sigma = 1 / (1 + np.exp(-x))

    dsigma = np.diagflat(sigma * (1 - sigma))
    rsigma = dsigma.T
    return sigma, rsigma


def foo(x):
    ## f: âĎİ^3 âĘŠ âĎİ^2 ##
    ## f(x, y, z) = (xy, z^2)

    y = np.zeros((2, 1))
    y[0] = x[0] * x[1]
    y[1] = x[2] ** 2

    J = np.zeros((2, 3))
```

```
    J[0, 0] = x[1]
    J[0, 1] = x[0]
    J[1, 2] = 2 * x[2]

    R = np.einsum("ij->ji", J)
    return y, R


def bar(x):
    ## f: âĎİ^{mÃŮn} âĘŠ âĎİ^m ##
    ## f(x) = x@v
    np.random.seed(1)
    m, n = x.shape
    v = np.random.randn(n)
    f = np.einsum("ij,␣j", x, v)

    J = np.zeros((m, m, n))
    for mu in range(m):
        for i in range(m):
            for j in range(n):
                if mu == i:
                    J[mu, i, j] = v[j]

    R = np.einsum("kij->ijk", J)
    return f, R


def baz(x):
    ## f: âĎİ^{mÃŮn} âĘŠ âĎİ^{mÃŮn} ##
    ## f(x) = x * x  # The Hadmard square
    m, n = x.shape
    f = np.einsum("ij,ij->ij", x, x)

    J = np.zeros((m, n, m, n))
    for mu in range(m):
        for nu in range(n):
            for i in range(m):
                for j in range(n):
                    if (mu == i) and (nu == j):
                        J[mu, nu, i, j] = 2 * x[i, j]

    R = np.einsum("ijkl->klij", J)
    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

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

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

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

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

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

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

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

iv. Perform gradient descent:

$$w^{[\ell]} := w^{[\ell]} - \alpha\frac{\partial \mathbb{J}^k}{\partial w^{[\ell]}}$$
$$b^{[\ell]} := b^{[\ell]} - \alpha\frac{\partial \mathbb{J}^k}{\partial b^{[\ell]}}$$

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.

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation

class ShuffleBatchData():
    def __init__(self, data, batch_size, seed=10101):
        """
        Parameters:
        -----------
        data : Dict[array_like]
            data['x'] : array_like
            data['y'] : array_like
        batch_size : int
        seed : int
            Default: 10101

        Returns:
        None
        """
        self.data = data
        self.batch_size = batch_size
        self.seed = seed
        self.idx = np.arange(data['x'].shape[1])
        self.__N = data['x'].shape[1]

        np.random.seed(seed)

    def get_batches(self):
        """
        Parameters:
        -----------
        None

        Returns:
        --------
        None
        """
        np.random.shuffle(self.idx)
        x_shuffled = self.data['x'][:, self.idx]
        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 reproducability
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):
```

```
        """
        Parameters:
        -----------
        params: Dict[LinearParameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            Default: 1e-8

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
        if self.lp_reg == 0:
            lambda_ = 0.0

        # Compute regularization term
        R = 0
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        R *= (lambda_ / (2 * n))

        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)    # Bound a for stability
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, params, cache, y, dropout):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

        Returns:
```

```
        --------
        None
        """

        # Retrieve cache
        a = cache['a']
        dg = cache['dg']

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * drop

    def update_parameters(self, params, learning_rate=0.1):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        learning_rate : float
            Default : 0.01

        Returns:
        --------
        None
        """
        for param in params.values():
            param.update(learning_rate)

    def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000):
        """
        Parameters:
        -----------
        data : Dict[array_like]
            data['x'] : array_like
            data['y'] : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

```

```
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

$$\begin{aligned}
v_t &= \beta_1 v_{t-1} + (1 - \beta_1) x_t \\
&= \beta_1(\beta_1 v_{t-2} + (1 - \beta_1) x_{t-1}) + (1 - \beta_1) x_t \\
&= \beta_1^2 v_{t-2} + (1 - \beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^2(\beta_1 v_{t-3} + (1 - \beta_1) x_{t-2}) + (1 - \beta_1)(\beta_1 x_{t-1} + x_t) \\
&= \beta_1^3 v_{t-3} + (1 - \beta_1)(\beta_1^2 x_{t-2} + \beta_1 x_{t-1} + x_t) \\
&\vdots \\
&= \beta_t^t v_0 + (1 - \beta_1) \sum_{j=0}^{t-1} \beta_1^j x_{t-j} \\
&= (1 - \beta_1) \sum_{j=0}^{t-1} \beta_1^j x_{t-j}.
\end{aligned}$$

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 $0 \leq i <$ `num_iters`:
   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$.
   b. For $1 \leq k \leq B$:
      i. Apply forward propagation on $\mathbb{X}^k$.
      ii. Compute the cost $\mathbb{J}$ on $\mathbb{X}^k$.
      iii. Apply backward propagation on $\mathbb{X}^k$ to obtain
      $$\frac{\partial \mathbb{J}}{\partial w}^{\{t\}}, \qquad \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}.$$
      iv. We update parameters
      $$w^{\{t\}} = w^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}}{\partial W}^{\{t\}}$$
      $$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}$$

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 $0 \leq i <$ `num_iters`:
   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$.
   b. For $1 \leq k \leq B$:

i. Apply forward propagation on $\mathbb{X}^k$.

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

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

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

iv. Define

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

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.

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation


class Momentum:
    def __init__(self, param, bias, beta1=0.9):
        """
        Parameters:
        -----------
        param : LinearParameters
        bias : Bool
        beta1 : float
            Default = 0.9

        Returns:
        --------
        None
```

```
        """
        self.bias = bias
        self.beta1 = beta1
        self.w = np.zeros(param.w.shape)
        if self.bias:
            self.b = np.zeros(param.b.shape)

    def update(self, param, learning_rate, iter, update_params=True):
        """
        Parameters:
        -----------
        param : LinearParameter
        learning_rate : float
        iter : int
        update_params : Bool
            Default = True  - Dictates return type

        Returns:
        --------
        None OR v : Dict[array_like]
        """
        self.w = self.beta1 * self.w + (1 - self.beta1) * param.dw
        vw_corrected = self.w / (1 - self.beta1**iter)
        if update_params:
            param.w = param.w - learning_rate * vw_corrected
        if self.bias:
            self.b = self.beta1 * self.b + (1 - self.beta1) * param.db
            vb_corrected = self.b / (1 - self.beta1**iter)
            if update_params:
                param.b = param.b - learning_rate * vb_corrected
        if not update_params:
            v = {}
            v["w"] = vw_corrected
            if self.bias:
                v["b"] = vb_corrected
            return v


class NeuralNetwork:
    def __init__(self, config):
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
            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 reproducability
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]]
```

```
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
        --------
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])
        if dropout != None:
            a[0] = dropout[0] * a[0]

        for l in range(1, self.L + 1):
            z = params[l].forward(a[l - 1])
            a[l], dg[l] = apply_activation(z, self.activators[l])
            if dropout != None:
                a[l] = dropout[l] * a[l]

        cache = {"a": a, "dg": dg}
        return cache

    def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params: Dict[LinearParameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            Default: 1e-8

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
        if self.lp_reg == 0:
```

```
            lambda_ = 0.0

        # Compute regularization term
        R = 0
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        R *= lambda_ / (2 * n)

        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)  # Bound a for stability
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, params, cache, y, dropout):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache["a"]
        dg = cache["dg"]

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = (
                dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
            )

```

```
    def update_parameters(self, params, moms, learning_rate, iter):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].b = Bias
        moms : Dict[Momentum]
        learning_rate : float
        iter : int

        Returns:
        --------
        None
        """
        for l in params.keys():
            moms[l].update(params[l], learning_rate, iter, True)

    def fit(
        self,
        data,
        learning_rate=0.1,
        lambda_=0.01,
        num_epochs=10000,
        print_cost_iter=1000,
    ):
        """
        Parameters:
        -----------
        data : Dict[array_like]
            data['x'] : array_like
            data['y'] : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000
        print_cost_iter : int
            Default: 1000   # 0 Doesn't print costs

        Returns:
        --------
        costs : List[floats]
        params : class[LinearParameters]
        """
        # Initialize parameters and optimizer per layer
```

```
        params = {}
        moms = {}
        for l in range(1, self.L + 1):
            params[l] = LinearParameters(
                (self.nodes[l], self.nodes[l - 1]), self.bias[l]
            )
            moms[l] = Momentum(params[l], self.bias[l])

        # Initialize batching
        batching = ShuffleBatchData(data, self.batch_size)

        costs = []
        for epoch in range(num_epochs):
            batches = batching.get_batches()
            B = len(batches)
            k = 1
            cost = 0
            for batch in batches:
                iter = (epoch * B) + k
                x = batch["x"]
                y = batch["y"]
                dropout = self.init_dropout(x.shape[1])
                cache = self.forward_propagation(params, x, dropout)
                batch_cost = self.cost_function(params, cache["a"][self.L], y, lambd
                cost += x.shape[1] * batch_cost
                self.backward_propagation(params, cache, y, dropout)
                self.update_parameters(params, moms, learning_rate, iter)
                k += 1
            cost /= data["x"].shape[1]
            costs.append(cost)

            if (print_cost_iter != 0) and (epoch % print_cost_iter == 0):
                print(f"Cost after epoch {epoch}: {cost}")

        return params, costs

    def evaluate(self, params, x):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
        x : array_like

        Returns:
        --------
        y_hat : array_like
        """
```

```
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 slowing 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

$$s \sim \frac{\partial \mathbb{J}}{\partial w}^2 ,$$

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 $0 \leq i <$ `num_iter`:

   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$

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

      i. Apply forward propagation on $\mathbb{X}^k$.

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

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

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

      iv. Define

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

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

      v. Update parameters via

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

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

7.3.1 Python Implementation via `numpy`

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation


class RMSProp:
    def __init__(self, param, bias, beta2=0.9, eps=1e-8):
        """
        Parameters:
        -----------
        params : LinearParameters
        bias : Bool
        beta2 : float
            Default = 0.9
        eps : float
            Default = 10^{-8}

        Returns:
        None
        """
        self.bias = bias
        self.beta2 = beta2
        self.eps = eps
        self.w = np.zeros(param.w.shape)
        if self.bias:
            self.b = np.zeros(param.b.shape)

    def update(self, param, learning_rate, iter, update_params=True):
        """
        Parameters:
        -----------
        params : LinearParameters
        learning_rate : float
        iter : int
        update_params : Boolean
            Default = True

        Returns:
        None OR v : Dict[array_like]
        """
        self.w = self.beta2 * self.w + (1 - self.beta2) * (param.dw**2)
        sw_corrected = self.w / (1 - self.beta2**iter)
```

```
        if update_params:
            param.w = param.w - learning_rate * (
                param.dw / (np.sqrt(sw_corrected) + self.eps)
            )
        if self.bias:
            self.b = self.beta2 * self.b + (1 - self.beta2) * (param.db**2)
            sb_corrected = self.b / (1 - self.beta2**iter)
            if update_params:
                param.b = param.b - learning_rate * (
                    param.db / (np.sqrt(sb_corrected) + self.eps)
                )
        if not update_params:
            s = {}
            s["w"] = sw_corrected
            if self.bias:
                s["b"] = sb_corrected
            return s


class NeuralNetwork:
    def __init__(self, config):
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
            config['nodes'] = List[int]
            config['bias'] = List[Boolean]
            config['activators'] = List[str]
            config['keep_probs'] = List[float]

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config["lp_reg"]
        self.batch_size = config["batch_size"]
        self.nodes = config["nodes"]
        self.bias = config["bias"]
        self.activators = config["activators"]
        self.keep_probs = config["keep_probs"]
        self.L = len(config["nodes"]) - 1

    def init_dropout(self, num_examples, seed=101011):
        """
```

```
        Parameters:
        -----------
        num_examples : int
        seed : int
            Default: 1 # For reproducability

        Returns:
        --------
        D : Dict[layer : array_like]
        """
        np.random.seed(seed)
        D = {}
        for l in range(self.L + 1):
            D[l] = np.random.rand(self.nodes[l], num_examples)
            D[l] = (D[l] < self.keep_probs[l]).astype(int)
            D[l] = D[l] / self.keep_probs[l]
            assert D[l].shape == (
                self.nodes[l],
                num_examples,
            ), "Dropout matrices are the wrong shape"

        return D

    def forward_propagation(self, params, x, dropout=None):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
        --------
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])
        if dropout != None:
            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=None):
184         """
185         Parameters:
```

```
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache["a"]
        dg = cache["dg"]

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = (a[self.L] - y) / y.shape[1]
        if dropout != None:
            delta[self.L] *= dropout[self.L]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1])
            if dropout != None:
                delta[l - 1] *= dropout[l - 1]

    def update_parameters(self, params, rmsprops, learning_rate, iter):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].b = Bias
        rmsprops : Dict[RMSProp]
        learning_rate : float
        iter : int

        Returns:
        --------
        None
        """
        for l in params.keys():
            rmsprops[l].update(params[l], learning_rate, iter, True)
```

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

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

```
        """
        x = data["x"]
        y = data["y"]

        y_hat = self.evaluate(params, x)
        acc = np.sum(y_hat == y) / y.shape[1]
```

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 it 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 $0 \leq i <$ `num_iters`:

   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$

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

      i. Apply forward propagation on $\mathbb{X}^k$.

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

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

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

      iv. Define

$$\begin{aligned} {v_w}^{\{t\}} &= \beta_1 {v_w}^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial w}^{\{t\}}, \\ {v_b}^{\{t\}} &= \beta_1 {v_b}^{\{t-1\}} + (1 - \beta_1) \frac{\partial \mathbb{J}}{\partial b}^{\{t\}}, \end{aligned}$$

and define

$$s_w{}^{\{t\}} = \beta_2 s_w{}^{\{t-1\}} + (1-\beta_2)\left(\frac{\partial \mathbb{J}}{\partial w}^{\{t\}}\right)^2,$$
$$s_b{}^{\{t\}} = \beta_2 s_b{}^{\{t-1\}} + (1-\beta_2)\left(\frac{\partial \mathbb{J}}{\partial b}^{\{t\}}\right)^2.$$

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

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

7.4.1 Python Implementation via `numpy`

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, Momentum, RMSProp
from mlLib.utils import apply_activation

```

```
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):
```

```
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
            config['nodes'] = List[int]
            config['bias'] = List[Boolean]
            config['activators'] = List[str]
            config['keep_probs'] = List[float]

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config["lp_reg"]
        self.batch_size = config["batch_size"]
        self.nodes = config["nodes"]
        self.bias = config["bias"]
        self.activators = config["activators"]
        self.keep_probs = config["keep_probs"]
        self.L = len(config["nodes"]) - 1

    def init_dropout(self, num_examples, seed=101011):
        """
        Parameters:
        -----------
        num_examples : int
        seed : int
            Default: 1 # For reproducability

        Returns:
        --------
        D : Dict[layer : array_like]
        """
        np.random.seed(seed)
        D = {}
        for l in range(self.L + 1):
            D[l] = np.random.rand(self.nodes[l], num_examples)
            D[l] = (D[l] < self.keep_probs[l]).astype(int)
            D[l] = D[l] / self.keep_probs[l]
            assert D[l].shape == (
                self.nodes[l],
                num_examples,
            ), "Dropout matrices are the wrong shape"

```

```
        return D

    def forward_propagation(self, params, x, dropout=None):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
        --------
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])
        if dropout != None:
            a[0] = dropout[0] * a[0]

        for l in range(1, self.L + 1):
            z = params[l].forward(a[l - 1])
            a[l], dg[l] = apply_activation(z, self.activators[l])
            if dropout != None:
                a[l] = dropout[l] * a[l]

        cache = {"a": a, "dg": dg}
        return cache

    def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params: Dict[LinearParameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            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 optimzer 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, lambd
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         -----------
```

```
        params : Dict[LinearParameters]
        x : array_like

        Returns:
        --------
        y_hat : array_like
        """
        cache = self.forward_propagation(params, x)
        a = cache["a"][self.L]
        y_hat = (~(a < 0.5)).astype(int)
        return y_hat

    def accuracy(self, params, data):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
        data : Dict[array_like]
            data['x'] : array_like
            data['y'] : array_like

        Returns:
        --------
        accuracy : float
        """
        x = data["x"]
        y = data["y"]

        y_hat = self.evaluate(params, x)
        acc = np.sum(y_hat == y) / y.shape[1]

        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`

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData
from mlLib.utils import apply_activation


def learning_rate_decay_rational(epoch, eta=1.0, alpha=0.2):
    """
    Parameters:
    -----------
    epoch : int
    eta : float
        Default = 1.0
    alpha : float
        Default = 0.2

    Returns:
    learning_rate : float
    """
    learning_rate = alpha / (1 + eta * epoch)
    assert (
        0 <= learning_rate <= 1
    ), f"learnining rate is outside [0,1] for epoch {epoch}"
    return learning_rate


def learning_rate_decay_exponential(epoch, eta=0.95, alpha=0.2):
    """
    Parameters:
    -----------
    epoch : int
    eta : float
        Default = 0.95
    alpha : float
        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
```

```
        """
        self.config = config
        self.lp_reg = config["lp_reg"]
        self.batch_size = config["batch_size"]
        self.nodes = config["nodes"]
        self.bias = config["bias"]
        self.activators = config["activators"]
        self.keep_probs = config["keep_probs"]
        self.L = len(config["nodes"]) - 1

    def init_dropout(self, num_examples, seed=101011):
        """
        Parameters:
        -----------
        num_examples : int
        seed : int
            Default: 1 # For reproducability

        Returns:
        --------
        D : Dict[layer : array_like]
        """
        np.random.seed(seed)
        D = {}
        for l in range(self.L + 1):
            D[l] = np.random.rand(self.nodes[l], num_examples)
            D[l] = (D[l] < self.keep_probs[l]).astype(int)
            D[l] = D[l] / self.keep_probs[l]
            assert D[l].shape == (
                self.nodes[l],
                num_examples,
            ), "Dropout matrices are the wrong shape"

        return D

    def forward_propagation(self, params, x, dropout=None):
        """
        Parameters:
        -----------
        params : Dict[class[Parameters]]
            params[l].w = Weights
            params[l].bias = Boolean
            params[l].b = Bias
        x : array_like

        Returns:
        --------
```

```
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg

        """
        # Initialize dictionaries
        a = {}
        dg = {}

        a[0], dg[0] = apply_activation(x, self.activators[0])
        if dropout != None:
            a[0] = dropout[0] * a[0]

        for l in range(1, self.L + 1):
            z = params[l].forward(a[l - 1])
            a[l], dg[l] = apply_activation(z, self.activators[l])
            if dropout != None:
                a[l] = dropout[l] * a[l]

        cache = {"a": a, "dg": dg}
        return cache

    def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
        """
        Parameters:
        -----------
        params: Dict[LinearParameters]
        a: array_like
        y: array_like
        lambda_: float
            Default: 0.01
        eps: float
            Default: 1e-8

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
        if self.lp_reg == 0:
            lambda_ = 0.0

        # Compute regularization term
        R = 0
        for param in params.values():
            R += np.sum(np.abs(param.w) ** self.lp_reg)
        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, lambd
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 (relabeled as):

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

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

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

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 it 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

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

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

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

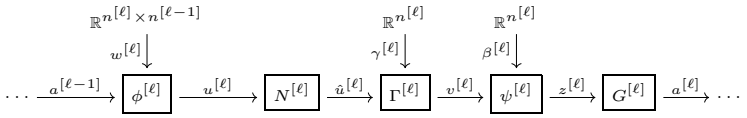
and so the new mean would be given by β. Similarly,

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

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 $BN_{\gamma^{[\ell]},\beta^{[\ell]}} : \mathbb{R}^{n^{[\ell]}} \to \mathbb{R}^{n^{[\ell]}}$ 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:



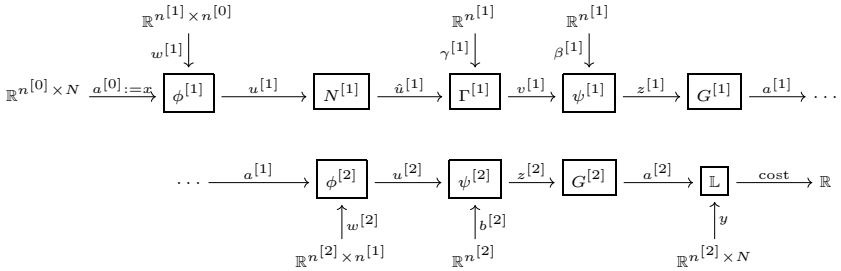
We note that we've dropped the bias term $b^{[\ell]}$ in the above, forward-propagating diagram. If we had included the term, the composition would result in the following

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

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

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

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

2.
$$G^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times N} \to \mathbb{R}^{n^{[\ell]} \times N}$$
is the broadcasting of the activation unit $g^{[\ell]} : \mathbb{R} \to \mathbb{R}$.

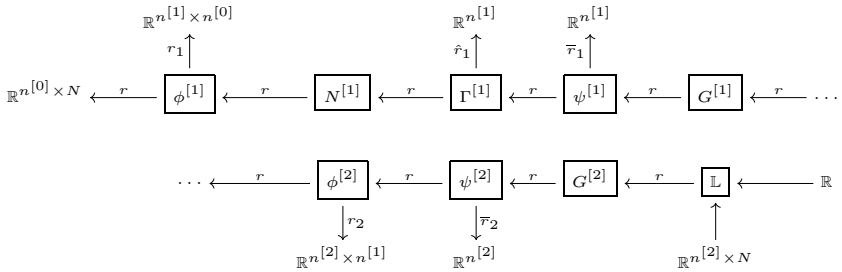
3.
$$\phi^{[\ell]} : \mathbb{R}^{n^{[\ell]} \times n^{[\ell-1]}} \times \mathbb{R}^{n^{[\ell-1]} \times N} \to \mathbb{R}^{n^{[\ell]} \times N}$$
is given by
$$\phi^{[\ell]}(w, x) = wx.$$

4.
$$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]} \times N} \to \mathbb{R}^{n^{[\ell]} \times N}$$
is given by
$$\psi(b, x) = x + b\vec{1}^T,$$
where
$$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^N.$$

5.
$$N^{[1]} : \mathbb{R}^{n^{[1]} \times N} \to \mathbb{R}^{n^{[1]} \times N}$$
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.
$$\Gamma^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]} \times N} \to \mathbb{R}^{n^{[\ell]} \times N}$$
is given by
$$\Gamma(\gamma, x) = \gamma\vec{1}^T \odot x,$$
where
$$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^N.$$

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



We consider our individual derivatives:

1. Suppose $G : \mathbb{R}^{m\times n} \to \mathbb{R}^{m\times n}$ is the broadcasting of $g : \mathbb{R} \to \mathbb{R}$. 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 $\phi : \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times N} \to \mathbb{R}^{m\times N}$ 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 $\psi : \mathbb{R}^n \times \mathbb{R}^{n\times N} \to \mathbb{R}^{n\times N}$ is given by

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

where

$$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^N.$$

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

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

with reverse differential

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

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

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

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

(a) In the network direction, we have that

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

with reverse differential

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

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

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

with reverse differential

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

5. For a coordinate-free derivation of the normalization operator, see Section B. Otherwise, we let

$$N : \mathbb{R}^{n\times N} \to \mathbb{R}^{n\times N}, \qquad y := N(x), \qquad y^\alpha_\beta = \frac{x^\alpha_\beta - \mu^\alpha}{\sqrt{\sigma^{2\alpha} + \epsilon}},$$

and note that

$$\frac{\partial \mu^\alpha}{\partial x^i_j} = \frac{1}{N}\delta^\alpha_i,$$

and

$$\frac{\partial \sigma^{2\alpha}}{\partial x^i_j} = \frac{2}{N}(x^\alpha_j - \mu^\alpha)\delta^\alpha_i.$$

Hence,

$$\begin{aligned}\frac{\partial y^\alpha_\beta}{\partial x^i_j} &= -\frac{1}{2}(\sigma^{2\alpha} + \epsilon)^{-\frac{3}{2}}\left(\frac{\partial \sigma^{2\alpha}}{\partial x^i_j}\right)(x^\alpha_\beta - \mu^\alpha) + (\sigma^{2\alpha} + \epsilon)^{-\frac{1}{2}}\left(\frac{\partial x^\alpha_\beta}{\partial x^i_j} - \frac{\partial \mu^\alpha}{\partial x^i_j}\right) \\ &= -\frac{1}{N}(\sigma^{2\alpha} + \epsilon)^{-\frac{3}{2}}(x^\alpha_j - \mu^\alpha)\delta^\alpha_i(x^\alpha_\beta - \mu^\alpha) + (\sigma^{2\alpha} + \epsilon)^{-\frac{1}{2}}\left(\delta^\alpha_i\delta^j_\beta - \frac{1}{N}\delta^\alpha_i\right) \\ &= -\frac{1}{N}(\sigma^{2\alpha} + \epsilon)^{-\frac{1}{2}}y^\alpha_j y^\alpha_\beta \delta^\alpha_i + (\sigma^{2\alpha} + \epsilon)^{-\frac{1}{2}}\left(\delta^j_\beta - \frac{1}{N}\right)\delta^\alpha_i \\ &= \frac{\delta^\alpha_i}{\sqrt{\sigma^{2\alpha} + \epsilon}}\left(\delta^j_\beta - \frac{1}{N} - \frac{1}{N}y^\alpha_j y^\alpha_\beta\right) \\ &= \frac{\delta^\alpha_i}{\sqrt{\sigma^{2\alpha} + \epsilon}}\left(\delta^j_\beta - \frac{1}{N}(1 + y^\alpha_j y^\alpha_\beta)\right).\end{aligned}$$

Thus for $(x, \xi^i_j) \in T\mathbb{R}^{n\times N}$, if we let $\mathcal{F}^\alpha{}_{\beta i}{}^j$ denote the rank $(2,2)$-tensor representation for the forward differential, we have that

$$\begin{aligned}dN_x(\xi) &= \frac{\partial y^\alpha_\beta}{\partial x^i_j}\xi^i_j \\ &= \mathcal{F}^\alpha{}_{\beta i}{}^j\xi^i_j,\end{aligned}$$

and for $\zeta_\beta^\alpha \in T_y\mathbb{R}^{n\times N}$, if we let $\mathcal{R}^i{}_{j\alpha}{}^\beta$ denote the rank $(2,2)$-tensor representation for the reverse differential, we have that

$$\begin{aligned} rN_x(\zeta) &= \sum_{\alpha=1}^{n}\sum_{\beta=1}^{N}\frac{\partial y_\beta^\alpha}{\partial x_j^i}\zeta_\beta^\alpha \\ &= \mathcal{R}^i{}_{j\alpha}{}^\beta\zeta_\beta^\alpha. \end{aligned}$$

6. For the loss function $\mathbb{L} : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ 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

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

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 $\alpha \in T_{\mathbb{J}(p)}\mathbb{R}$ with the assumption that our second activator function is the sigmoid function. Indeed,

$$
\begin{aligned}
r(\mathbb{L} \circ G^{[2]})_{z^{[2]}}(\alpha) &= rG^{[2]}_{z^{[2]}} \circ r\mathbb{L}_{a^{[2]}}(\alpha) \\
&= -\frac{\alpha}{N} G^{[2]\prime}(z^{[2]}) \odot \left[\frac{y}{a^{[2]}} - \frac{1-y}{1-a^{[2]}} \right] \\
&= -\frac{\alpha}{N} a^{[2]}(1-a^{[2]}) \left[\frac{y}{a^{[2]}} - \frac{1-y}{1-a^{[2]}} \right] \\
&= -\frac{\alpha}{N} [y(1-a^{[2]}) - a^{[2]}(1-y)] \\
&= -\frac{\alpha}{N} [y - a^{[2]}] \\
&= \frac{\alpha}{N} (a^{[2]} - y).
\end{aligned}
$$

This leads us to

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

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

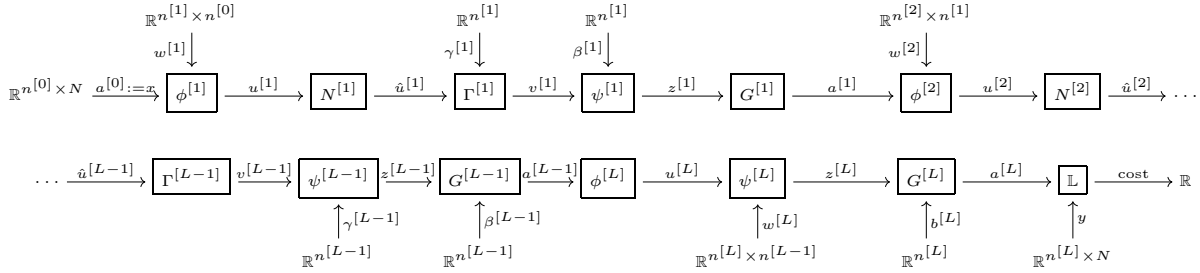
$$
\begin{aligned}
\overline{r}_1 \mathbb{J}_p(\alpha) &= \overline{r}_1 \psi^{[1]}_{(\beta^{[1]}, v^{[1]})} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{(w^{[2]}, a^{[2]})} \circ r\psi^{[2]}_{(b^{[2]}, u^{[2]}} \circ r(\mathbb{L} \circ G^{[2]})_{z^{[2]}}(\alpha) \\
&= \frac{\alpha}{N} \overline{r}_1 \psi^{[1]}_{(\beta^{[1]}, v^{[1]})} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{(w^{[2]}, a^{[2]})} (a^{[2]} - y) \\
&= \frac{\alpha}{N} \overline{r}_1 \psi^{[1]}_{(\beta^{[1]}, v^{[1]})} \circ rG^{[1]}_{z^{[1]}} \left(w^{[2]T} (a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} \overline{r}_1 \psi^{[1]}_{(\beta^{[1]}, v^{[1]})} \left(G^{[1]\prime}(z^{[1]}) \odot w^{[2]T} (a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} \sum_{j=1}^{N} g^{[1]\prime}({z^{[1]}}_j) \odot w^{[2]T} ({a^{[2]}}_j - y_j);
\end{aligned}
$$

$$\begin{aligned}
\hat{r}_1\mathbb{J}_p(\alpha) &= \frac{\alpha}{N}\hat{r}_1\Gamma^{[1]}_{(\gamma^{[1]},\hat{u}^{[1]})}\left(G^{[1]\prime}(z^{[1]})\odot w^{[2]T}(a^{[2]}-y)\right)\\
&= \frac{\alpha}{N}R_{\vec{1}}\left(\hat{u}^{[1]}\odot\left(G^{[1]\prime}(z^{[1]})\odot w^{[2]T}(a^{[2]}-y)\right)\right)\\
&= \frac{\alpha}{N}\sum_{j=1}^{n}\hat{u}^{[1]}{}_j\odot g^{[1]\prime}(z^{[1]}{}_j)\odot w^{[2]T}(a^{[2]}{}_j-y_j);
\end{aligned}$$

and finally,

$$\begin{aligned}
r_1\mathbb{J}_p(\alpha) &= \frac{\alpha}{N}r_1\phi^{[1]}_{(w^{[1]},x)}\circ rN^{[1]}_{u^{[1]}}\circ r\Gamma^{[1]}_{(\gamma^{[1]},\hat{u}^{[1]})}\left(G^{[1]\prime}(z^{[1]})\odot w^{[2]T}(a^{[2]}-y)\right)\\
&= \frac{\alpha}{N}r_1\phi^{[1]}_{(w^{[1]},x)}\circ rN^{[1]}_{u^{[1]}}\left(\gamma\vec{1}^T\odot G^{[1]\prime}(z^{[1]})\odot w^{[2]T}(a^{[2]}-y)\right)\\
&= \frac{\alpha}{N}R_{x^T}\circ rN^{[1]}_{u^{[1]}}\left(\gamma\vec{1}^T\odot G^{[1]\prime}(z^{[1]})\odot w^{[2]T}(a^{[2]}-y)\right)\\
&= \frac{\alpha}{N}\sum_{j,l=1}^{N}\sum_{i=1}^{n^{[1]}}\mathcal{R}^k{}_{li}{}^j\gamma^i g^{[1]\prime}(z^{[1]i}{}_j)w^{[2]}{}_i(a^{[2]}{}_j-y_j)x^m_l.
\end{aligned}$$

In general, to simply the construction in python, we utilize the auxiliary δ's as before. To this, suppose we have an arbitrary L-layer neural network utilizing batch normalization on all hidden layers as in the following diagram:



Then we build our δ-differentials recursively starting at $\mathbb{L}$. That is,

$$\begin{aligned}
\delta^{[L]} &:= r(\mathbb{L} \circ G^{[L]})_{z^{[L]}}, \\
\delta^{[L-1]} &:= G^{[L-1]\prime}(z^{[L-1]}) \odot w^{[L]T} \delta^{[L]}, \\
\delta^{[L-2]} &:= G^{[L-2]\prime}(z^{[L-2]}) \odot w^{[L-1]T} r N^{[L-1]}_{u^{[L-1]}}(\delta^{[L-1]}), \\
&\vdots \\
\delta^{[\ell]} &:= G^{[\ell]\prime}(z^{[\ell]}) \odot w^{[\ell+1]T} r N^{[\ell+1]}_{u^{[\ell+1]}}(\delta^{[\ell+1]}), \\
&\vdots \\
\delta^{[1]} &:= G^{[1]\prime}(z^{[1]}) \odot w^{[2]T} r N^{[2]}_{u^{[2]}}(\delta^{[2]});
\end{aligned}$$

and compute our gradients via

$$\begin{aligned}
\frac{\partial \mathbb{J}}{\partial b^{[L]}} &= \sum_{j=1}^{N} {\delta^{[L]}}_j, \\
\frac{\partial \mathbb{J}}{\partial w^{[L]}} &= \delta^{[L]} a^{[L-1]T}, \\
\frac{\partial \mathbb{J}}{\partial \beta^{[\ell]}} &= \sum_{j=1}^{N} \delta^{[\ell]}, \qquad \ell \in \{1, ..., L-1\}, \\
\frac{\partial \mathbb{J}}{\partial \gamma^{[\ell]}} &= \sum_{j=1}^{N} {\hat{u}^{[\ell]}}_j \odot {\delta^{[\ell]}}_j, \qquad \ell \in \{1, ..., L-1\}, \\
\frac{\partial \mathbb{J}}{\partial w^{[\ell]}} &= r N^{[\ell]}_{u^{[\ell]}}(\delta^{[\ell]}) a^{[\ell-1]T}, \qquad \ell \in \{1, ..., L-1\}.
\end{aligned}$$

9.2 Inferencing

We note that in our computation for forward propagation, that our normalization transforms change with our batches. This leads to ambiguity when predicting a label for a new example. To fix this ambiguity, we will use exponentially moving averages to accumulate the means and variances of a given layer. That is, given a mini-batch partition $\{\mathbb{X}^k : 1 \leq k \leq B\}$ and epoch i, we let

$$t := iB + k,$$

denote the iteration. Fixing a layer ℓ, we let

$$\overline{\mu}^{[\ell]}{}_0 = 0, \qquad \overline{\sigma}^{2[\ell]}{}_0 = 0,$$

and for some momentum $\beta \in [0, 1]$ we define

$$\overline{\mu}^{[\ell]}{}_t = \beta \overline{\mu}^{[\ell]}{}_{t-1} + (1 - \beta)\mu^{[\ell]}{}_t,$$

and

$$\overline{\sigma}^{2[\ell]}{}_t = \beta \overline{\sigma}^{2[\ell]}{}_{t-1} + (1 - \beta)\sigma^{2[\ell]}{}_t.$$

After convergence, we use $\overline{\mu}^{[\ell]}$ and $\overline{\sigma}^{2[\ell]}$ in evaluation on our tests sets.

9.3 Algorithm Outline

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

1. Set hyper-parameters. Initialize parameters and running statistics.
2. For $0 \leq i \leq$ num_epochs:
   a. Generate batches $\{\mathbb{X}^k : 1 \leq k \leq B\}$.
   b. For $1 \leq k \leq B$:
      i. $$t = iB + k.$$
      ii. Perform forward propagation on $\mathbb{X}^k$:
      - $$u^{[1]} = w^{[1]}x$$
      - For $\ell \in \{1, ..., L - 1\}$:
        – $$u^{[\ell]} = w^{[\ell]}a^{[\ell-1]}$$
        – $$\mu^{[\ell]}{}_t = \frac{1}{n}\sum_{x \in \mathbb{X}^k} u^{[\ell]}$$
        – $$\overline{\mu}^{[\ell]}{}_t = \beta \overline{\mu}^{[\ell]}{}_{t-1} + (1 - \beta)\mu^{[\ell]}{}_t$$
        – $$\sigma^{2[\ell]}{}_t = \frac{1}{n}\sum_{x \in \mathbb{X}^k} (u^{[\ell]} - \mu^{[\ell]}{}_t)^2$$

– $$\overline{\sigma}^{2^{[\ell]}}{}_t = \beta\overline{\sigma}^{2^{[\ell]}}{}_{t-1} + (1-\beta)\sigma^{2^{[\ell]}}{}_t$$

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

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

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

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

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

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

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

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

v. Update parameters.

3. Return

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

9.4 Python Implementation via `numpy`

Below we implement batch normalization with our usual mini-batch gradient descent optimization. To adapt other optimization techniques an update to the optimization classes is required to account for the trainable parameters. This is not difficult, but is not included in this implementation for brevity.

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, EpochRuntime
from mlLib.utils import apply_activation


class BatchNormParameters:
    def __init__(self, dim, eps=1e-8):
```

```
        """
        Parameters:
        -----------
        dim = int
        eps : float
            Default = 10^{-8}

        Returns:
        --------
        None
        """
        self.dims = (dim, 1)
        self.eps = eps
        self.gamma = np.ones(self.dims)
        self.beta = np.zeros(self.dims)

        self.running_mean = np.zeros(self.dims)
        self.running_var = np.zeros(self.dims)

    def normalize(self, u):
        """
        Parameters:
        -----------
        u : array_like
            u.shape == (n, N)
        iter : int

        Returns:
        uhat : array_like
            uhat.hape == (n, N)
        ruhat : array_like
            ruhat.shape == (n, N, n, N)
        """
        # Compute normalization
        mu = np.mean(u, axis=1, keepdims=True)
        sigma2 = np.var(u, axis=1, keepdims=True)
        theta = 1 / np.sqrt(sigma2 + self.eps)
        uhat = theta * (u - mu)

        # Update running mean and variance
        momentum = 0.9
        self.running_mean = momentum * self.running_mean + (1 - momentum) * mu
        self.running_var = momentum * self.running_var + (1 - momentum) * sigma2

        # Compute reverse differential
        m, n = u.shape
        duhat = np.zeros((m, n, m, n))
```

```
        I_m = np.eye(m)
        I_n = np.eye(n)
        for alpha in range(m):
            for beta in range(n):
                for i in range(m):
                    for j in range(n):
                        duhat[alpha, beta, i, j] = (
                            I_m[alpha, i]
                            * theta[alpha, 0]
                            * (
                                I_n[j, beta]
                                - (1 + uhat[alpha, j] * uhat[alpha, beta]) / n
                            )
                        )

        ruhat = np.einsum("ijkl->klij", duhat)

        return uhat, ruhat

    def forward(self, u):
        """
        Parameters:
        -----------
        u : array_like
            u.shape == (n, N)

        Returns:
        z : array_like
            z.shape == (n, N)
        """
        self.norm, self.dnorm = self.normalize(u)
        z = self.gamma * self.norm + self.beta
        return z

    def backward(self, d_in):
        """
        Parameters:
        -----------
        d_in : array_like
            d_in.shape == (n, N)
        """
        self.dbeta = np.sum(d_in, axis=1, keepdims=True)
        self.dgamma = np.sum(self.norm * d_in, axis=1, keepdims=True)

        return np.einsum("ijkl,kl", self.dnorm, d_in)

    def update(self, learning_rate):
```

```
        """
        Parameters:
        -----------
        learning_rate : float

        Returns:
        --------
        None
        """
        self.gamma = self.gamma - learning_rate * self.dgamma
        self.beta = self.beta - learning_rate * self.dbeta

    def evaluate(self, u):
        """
        Parameters:
        -----------
        u : array_like
            u.shape == (n, N)

        Returns:
        z : array_like
            z.shape == (n, N)
        """
        z = (u - self.running_mean) / np.sqrt(self.running_var + self.eps)
        z = self.gamma * z + self.beta
        return z


class NeuralNetwork:
    def __init__(self, config):
        """
        Parameters:
        -----------
        config : Dict
            config['lp_reg'] = 0,1,2
            config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
            config['nodes'] = List[int]
            config['bias'] = List[Bool]
            config['batch_norm'] = List[Bool]
            config['activators'] = List[str]
            config['keep_probs'] = List[float]

        Returns:
        --------
        None
        """
        self.config = config
```

```
        self.lp_reg = config["lp_reg"]
        self.batch_size = config["batch_size"]
        self.nodes = config["nodes"]
        self.bias = config["bias"]
        self.batch_norm = config["batch_norm"]
        self.activators = config["activators"]
        self.keep_probs = config["keep_probs"]
        self.L = len(config["nodes"]) - 1

    def init_dropout(self, num_examples, seed=101011):
        """
        Parameters:
        -----------
        num_examples : int
        seed : int
            Default: 1 # For reproducability

        Returns:
        --------
        D : Dict[layer : array_like]
        """
        np.random.seed(seed)
        D = {}
        for l in range(self.L + 1):
            D[l] = np.random.rand(self.nodes[l], num_examples)
            D[l] = (D[l] < self.keep_probs[l]).astype(int)
            D[l] = D[l] / self.keep_probs[l]
            assert D[l].shape == (
                self.nodes[l],
                num_examples,
            ), "Dropout matrices are the wrong shape"

        return D

    def forward_propagation(self, x, dropout=None):
        """
        Parameters:
        -----------
        x : array_like
        dropout : Dict[array_like]
            Default = None

        Returns:
        --------
        cache = Dict[array_like]
            cache['a'] = a
            cache['dg'] = dg
```

```
199
200         """
201         # Initialize dictionaries
202         a = {}
203         dg = {}
204
205         a[0], dg[0] = apply_activation(x, self.activators[0])
206         if dropout != None:
207             a[0] *= dropout[0]
208
209         for l in range(1, self.L + 1):
210             z = self.lin_params[l].forward(a[l - 1])
211             if self.batch_norm[l]:
212                 z = self.bn_params[l].forward(z)
213             a[l], dg[l] = apply_activation(z, self.activators[l])
214             if dropout != None:
215                 a[l] *= dropout[l]
216
217         cache = {"a": a, "dg": dg}
218         return cache
219
220     def cost_function(self, a, y, lambda_=0.01, eps=1e-8):
221         """
222         Parameters:
223         -----------
224         params: Dict[LinearParameters]
225         a: array_like
226         y: array_like
227         lambda_: float
228             Default: 0.01
229         eps: float
230             Default: 1e-8
231
232         Returns:
233         --------
234         cost: float
235         """
236         n = y.shape[1]
237         if self.lp_reg == 0:
238             lambda_ = 0.0
239
240         # Compute regularization term
241         R = 0
242         for param in self.lin_params.values():
243             R += np.sum(np.abs(param.w) ** self.lp_reg)
244         R *= lambda_ / (2 * n)
245
```

```
        # Compute unregularized cost
        a = np.clip(a, eps, 1 - eps)  # Bound a for stability
        J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))

        cost = float(np.squeeze(J + R))

        return cost

    def backward_propagation(self, cache, y, dropout):
        """
        Parameters:
        -----------
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like
        dropout : Dict[array_like]

        Returns:
        --------
        None
        """

        # Retrieve cache
        a = cache["a"]
        dg = cache["dg"]

        # Initialize differentials along the network
        delta = {}
        delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]

        for l in reversed(range(1, self.L + 1)):
            delta[l - 1] = (
                dg[l - 1]
                * self.lin_params[l].backward(delta[l], a[l - 1])
                * dropout[l - 1]
            )
            if self.batch_norm[l - 1]:
                delta[l - 1] = self.bn_params[l - 1].backward(delta[l - 1])

    def update_parameters(self, learning_rate):
        """
        Parameters:
        -----------
        learning_rate : float

        Returns:
```

```
293         --------
294         None
295         """
296         for l in range(1, self.L + 1):
297             self.lin_params[l].update(learning_rate)
298             if self.batch_norm[l]:
299                 self.bn_params[l].update(learning_rate)
300 
301     def fit(
302         self,
303         data,
304         learning_rate,
305         lambda_=0.01,
306         num_epochs=10000,
307         print_cost_epoch=1000,
308     ):
309         """
310         Parameters:
311         -----------
312         data : Dict[array_like]
313             data['x'] : array_like
314             data['y'] : array_like
315         learning_rate : float
316         lambda_ : float
317             Default = 0.01
318         num_epochs : int
319             Default = 10000
320         print_cost_epoch : int
321             Default = 1000   # 0 Doesn't print costs
322 
323         Returns:
324         --------
325         costs : List[floats]
326         params : Dict[LinearParameters]
327         """
328         # Initialize parameters per layer
329         self.lin_params = {}
330         self.bn_params = {}
331         for l in range(1, self.L + 1):
332             self.lin_params[l] = LinearParameters(
333                 (self.nodes[l], self.nodes[l - 1]), self.bias[l]
334             )
335             if self.batch_norm[l]:
336                 self.bn_params[l] = BatchNormParameters(self.nodes[l])
337 
338         # Initialize batching
339         batching = ShuffleBatchData(data, self.batch_size)
```

```
340
341         costs = []
342         time = EpochRuntime()
343         for epoch in range(num_epochs):
344             batches = batching.get_batches()
345             B = len(batches)
346             k = 1
347             cost = 0
348             for batch in batches:
349                 iter = epoch * B + k
350                 x = batch["x"]
351                 y = batch["y"]
352                 dropout = self.init_dropout(x.shape[1])
353                 cache = self.forward_propagation(x, dropout)
354                 batch_cost = self.cost_function(cache["a"][self.L], y, lambda_)
355                 cost += x.shape[1] * batch_cost
356                 self.backward_propagation(cache, y, dropout)
357                 self.update_parameters(learning_rate)
358                 k += 1
359             cost /= data["x"].shape[1]
360             costs.append(cost)
361             time.elapsed_time()
362
363             if (print_cost_epoch != 0) and (epoch % print_cost_epoch == 0):
364                 print(f"Cost␣after␣epoch␣{epoch}:␣{cost}")
365
366         return costs
367
368     def evaluate(self, x):
369         """
370         Parameters:
371         -----------
372         x : array_like
373
374         Returns:
375         --------
376         y_hat : array_like
377         """
378         a = {}
379         a[0], _ = apply_activation(x, self.activators[0])
380         for l in range(1, self.L + 1):
381             z = self.lin_params[l].forward(a[l - 1])
382             if self.batch_norm[l]:
383                 z = self.bn_params[l].evaluate(z)
384             a[l], _ = apply_activation(z, self.activators[l])
385
386         y_hat = (~(a[self.L] < 0.5)).astype(int)
```

```
387         return y_hat
388
389     def accuracy(self, data):
390         """
391         Parameters:
392         -----------
393         data : Dict[array_like]
394             data['x'] : array_like
395             data['y'] : array_like
396
397         Returns:
398         --------
399         accuracy : float
400         """
401         x = data["x"]
402         y = data["y"]
403
404         y_hat = self.evaluate(x)
405         acc = np.sum(y_hat == y) / y.shape[1]
406
407         return acc
```

10 Multi-Class Softmax Regression

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

Table 1: Classification

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

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

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

where C is the number of labels in our classification.

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

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

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

$$\{e_1, ..., e_C\},$$

which clearly admits a bijection

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

Thus, we've effectively mapped our true labels

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

where

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

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

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

then we can declare a prediction via

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

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

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

and

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

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

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

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

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

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

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

where

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

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

To minimize our cost, we first note

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

then we see that

$$\begin{aligned}
\frac{\partial z^\mu}{\partial W^\alpha_\beta} &= \frac{\partial}{\partial W^\alpha_\beta} (W^\mu_k x^k + b^\mu) \\
&= \sum_{k=1}^{m} \delta^\mu_\alpha \delta^\beta_k x^k \\
&= \delta^\mu_\alpha x^\beta,
\end{aligned}$$

and

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

Hence,

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

yielding a gradient of

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

and similarly

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

and so

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

Finally, we conclude that

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

and

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

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

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 $f : M \to N$ is smooth. Then for $p \in M$, the (exterior) differential of f at p, denoted df_p, is the linear map*

$$df_p : T_pM \to T_{f(p)}N$$

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

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

Example A.2. *Suppose $f : \mathbb{R}^n \to \mathbb{R}^m$ 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*

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

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

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

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

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

It then follows that

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

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

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

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 $f : \mathbb{R}^n \to \mathbb{R}^m$ 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*

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

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

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

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

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

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 $f := \psi \circ \phi : (M, g) \to (Q, k)$*. 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 $\xi \in T_pM$ 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* $f : (\mathbb{R}^{m\times n}, (X^i_j), F) \to (\mathbb{R}, (t), \delta)$ *is smooth, where* F *is the Frobenius inner product. Suppose* $v \in T_P\mathbb{R}^{m\times n} \leftrightsquigarrow V \in \mathbb{R}^{m\times n}$ *are represented via*

$$v = v^i_j \left.\frac{\partial}{\partial X^i_j}\right|_P \leftrightsquigarrow V = \left[v^i_j\right],$$

and in coordinates, we have that

$$df_P = \left.\frac{\partial f}{\partial X^i_j}\right|_P \left. dX^i_j\right|_P.$$

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 $\tau \in \mathbb{R} \leftrightsquigarrow \vec{\tau} = \tau \left.\frac{d}{dt}\right|_{f(P)} \in T_{f(P)}\mathbb{R}$, 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 $f : \mathbb{R}^{n\times m} \to \mathbb{R}^k$, and for $P \in \mathbb{R}^{n\times m}$, let $R = rf_P$. Then $R \in \mathbb{R}^k{}_n{}^m$ 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 $\frac{\partial}{\partial X^\nu_\mu} \in T_P\mathbb{R}^{n\times m}$ and $\frac{\partial}{\partial x^i} \in T_{f(P)}\mathbb{R}^k$ we have that

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

as desired. □

B The Normalization Operator

sec:normOp

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

$$N(x_1, ..., x_n) = (y_1, ..., y_n),$$

where

$$y_\beta = \frac{x_\beta - \mathbb{E}[x]}{\sqrt{\mathbb{V}[x] + \epsilon}}.$$

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 $x \in (\mathbb{R}^m)^n$, $j, \beta \in \{1, ..., n\}$, and let $\xi \in T_{x_j}\mathbb{R}^m$ and let

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

so we may consider

$$d_j y_x(\xi).$$

To this end, if we let

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

and consider y written compositionally as

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

then by the chain rule it follows that

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

Computing these differentials yields

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

Substituting in these differentials, we see that

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

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

$$r_j(y_\beta)_x \cong d_j(y_\beta)_x,$$

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

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

Moreover, since

$$d(y_\beta)_x(\xi) = \sum_{j=1}^{n} d_j(y_\beta)_x(\xi_j), \qquad \xi_\alpha \in T_{x_j}\mathbb{R}^m,$$

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

$$\begin{aligned}
\langle r(y_\beta)_x(\zeta_\beta), \xi\rangle_{(\mathbb{R}^m)^n} &= \langle \zeta_\beta, d(y_\beta)_x(\xi)\rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \left\langle \zeta_\beta, \sum_{j=1}^{n} d_j(y_\beta)_x(\xi_j) \right\rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{j=1}^{n} \langle r_j(y_\beta)(\zeta_\beta), \xi_j\rangle_{T_{x_j}\mathbb{R}^m} \\
&= \left\langle \bigoplus_{j=1}^{n} r_j(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n},
\end{aligned}$$

and hence that

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

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

$$\begin{aligned}
\langle rN_x(\zeta), \xi \rangle_{(\mathbb{R}^m)^n} &= \langle \zeta, dN_x(\xi) \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \zeta, \bigoplus_{\beta=1}^{n} d(y_\beta)_x(\xi) \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^{n} \langle \zeta_\beta, d(y_\beta)_x(\xi) \rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \sum_{j=1}^{n} \langle \zeta_\beta, d_j(y_\beta)_x(\xi_j) \rangle_{T_{y_\beta}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \sum_{j=1}^{n} \langle r_j(y_\beta)_x(\zeta_\beta), \xi_j \rangle_{T_{x_j}\mathbb{R}^m} \\
&= \sum_{\beta=1}^{n} \left\langle \bigoplus_{j=1}^{n} r_j(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n} \\
&= \sum_{\beta=1}^{n} \langle r(y_\beta)_x(\zeta_\beta), \xi \rangle_{(\mathbb{R}^m)^n} \\
&= \left\langle \sum_{\beta=1}^{n} r(y_\beta)_x(\zeta_\beta), \xi \right\rangle_{(\mathbb{R}^m)^n}.
\end{aligned}$$

That is,

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

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 $\{E_\alpha^\beta\}$ denote the basis for $\mathbb{R}^{m\times n}$, where

$$(E_\alpha^\beta)_l^k = \delta_\alpha^k \delta_l^\beta,$$

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

$$\begin{aligned}
rN_x(E_\alpha^\beta) &= \bigoplus_{j=1}^{n} \left\{ \theta \odot (E_\alpha^\beta e_j) - \frac{1}{n}\theta \odot (E_\alpha^\beta \vec{1}) - \frac{1}{n}\theta \odot y_l \odot (y \odot E_\alpha^\beta)\vec{1} \right\} \\
&= \bigoplus_{j=1}^{n} \left\{ \theta^i \delta_\alpha^i \delta_j^\beta \epsilon_i - \frac{1}{n}\theta^i \delta_\alpha^i \epsilon_i - \frac{1}{n}\theta^i y_j^i y_\beta^i \delta_\alpha^i \epsilon_i \right\} \\
&= \bigoplus_{j=1}^{n} \theta^i \left\{ \delta_\alpha^i \delta_j^\beta - \frac{1}{n}\delta_\alpha^i (1 + y_j^i y_\beta^i) \right\} \epsilon_i \\
&= \theta^i \left[\delta_\alpha^i \delta_j^\beta - \frac{1}{n}\delta_\alpha^i (1 + y_j^i y_\beta^i) \right] E_i^j \qquad \text{definition of direct sum,}
\end{aligned}$$

that is, if ζ_β^α is a matrix, we yield the matrix

$$[rN_x(\zeta)]_j^i = \sum_{\alpha=1}^{m}\sum_{\beta=1}^{n} \theta^i [\delta_\alpha^i \delta_j^\beta + \frac{1}{n}\delta_\alpha^i (1 + y_j^i y_\beta^i)]\zeta_\beta^\alpha,$$

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 Shiebler, 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.