Neural Networks

Matt R

July 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

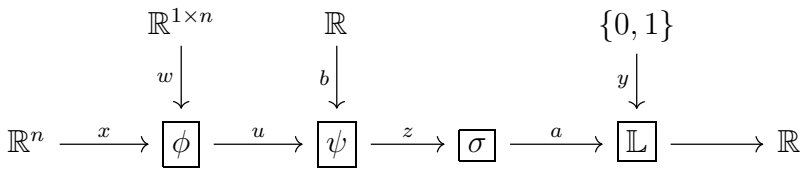
We wish to compute the gradient of our cost function $\mathbb{J}$ with respect to our trainable parameters, $w \in \mathbb{R}^{1\times n}$ and $b \in \mathbb{R}$. To this end, we define the functions

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

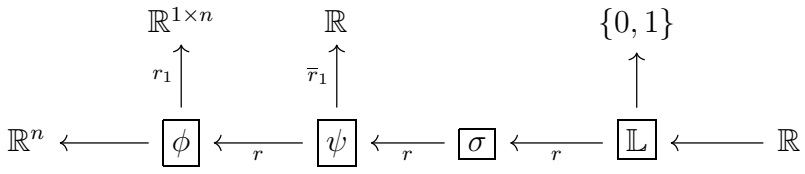
and

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

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          return z
44
45      def backward(self, dz, x):
46          """
47          Parameters:
48          -----------
49          dz : array_like
50          x : array_like
51
52          Returns:
53          --------
54          None
55          """
56          if self.bias:
57              self.db = np.sum(dz, axis=1, keepdims=True)
58              assert (self.db.shape == self.b.shape)
59
60          self.dw = np.einsum('ij,kj', dz, x)
61          assert (self.dw.shape == self.w.shape)
62
63      def update(self, learning_rate=0.01):
64          """
65          Parameters:
66          -----------
67          learning_rate : float
68              Default : 0.01
69
70          Returns:
71          --------
72          None
73          """
74          w = self.w - learning_rate * self.dw
75          self.w = w
76
77          if self.bias:
78              b = self.b - learning_rate * self.db
79              self.b = b
80
81  class LogisticRegression():
82      def __init__(self, lp_reg):
83          """
84          Parameters:
85          lp_reg : int
86              2 : L_2 Regularization is imposed
87              1 : L_1 Regularization is imposed
88              0 : No regulariation is imposed
89
```

```
90          Returns:
91          --------
92          None
93          """
94          self.lp_reg = lp_reg
95
96      def predict(self, params, x):
97          """
98          Parameters:
99          -----------
100         params : class[LinearParameters]
101         x : array_like
102
103         Returns:
104         --------
105         a : array_like
106         dg : array_like
107         """
108         z = params.forward(x)
109         a, dg = apply_activation(z, 'sigmoid')
110         return a, dg
111
112     def cost_function(self, params, x, y, lambda_=0.01, eps=1e-8):
113         """
114         Parameters:
115         -----------
116         params : class[LinearParameters]
117         x : array_like
118         y : array_like
119         lambda_ : float
120             Default : 0.01
121         eps : float
122             Default : 1e-8
123
124         Returns:
125         --------
126         cost : float
127         """
128         n = y.shape[1]
129
130         R = np.sum(np.abs(params.w) ** self.lp_reg)
131         R *= (lambda_ / (2 * n))
132
133         a, _ = self.predict(params, x)
134         a = np.clip(a, eps, 1 - eps)
135
136         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
```

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

```
        -----------
        params : class[Parameters]
        x : array_like

        Returns:
        --------
        y_hat : array_like
        """
        a, _ = self.predict(params, x)
        y_hat = (~(a < 0.5)).astype(int)

        return y_hat

    def accuracy(self, params, x, y):
        """
        Parameters:
        -----------
        params : class[Parameters]
        x : array_like
        y : array_like

        Returns:
        --------
        accuracy : float
        """
        y_hat = self.evaluate(params, x)

        accuracy = np.sum(y_hat == y) / y.shape[1]

        return accuracy
```

1.3 Implementation in Python via `sklearn`

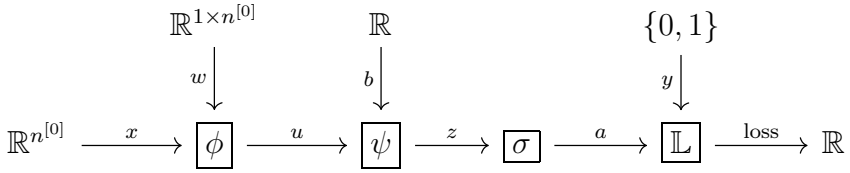
Here we include the general method of coding a logistic regression model via `scikit-learn`'s modeling library.

```
#! python3

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

def main(csv):
    df = pd.read_csv(csv)
    dataset = df.values
```

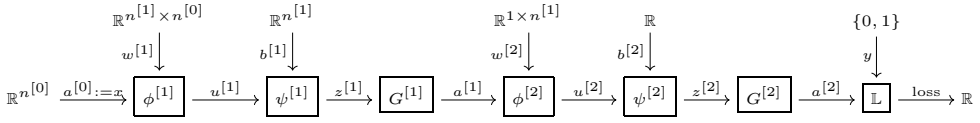
```
    x = dataset[:, :10]
    y = dataset[:, 10]

    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

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

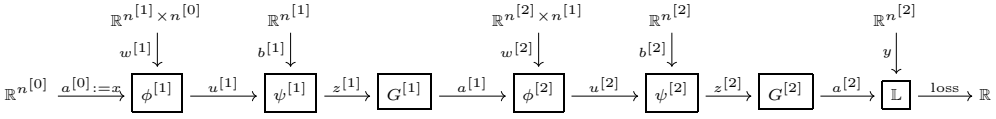
That is, as a map $dS_z : T_z\mathbb{R}^m \to T_{S(z)}\mathbb{R}^m$, we have that

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

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

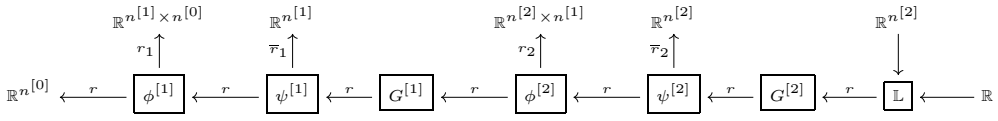
   is given by

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

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. $r_1\mathbb{J}_p$:

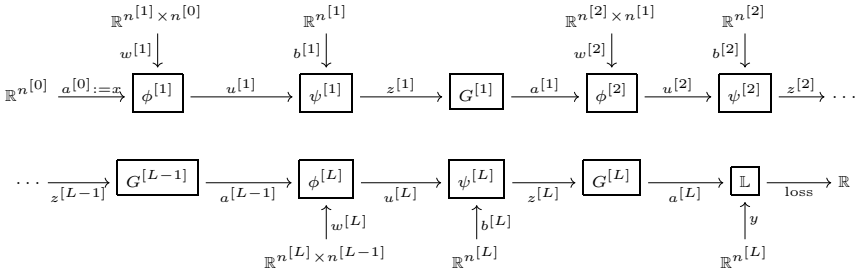
$$\begin{aligned} r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1\phi^{[1]}_p \circ r\psi^{[1]}_{u^{[1]}} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{a^{[1]}} \circ r\psi^{[2]}_{u^{[2]}} \circ rG^{[2]}_{z^{[2]}} \circ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

and hence

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

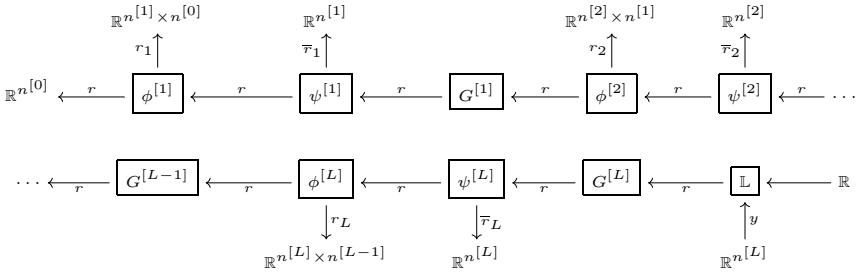
3 Deep Neural Networks

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)))
```

```
88
89          cost = float(np.squeeze(J + R))
90
91          return cost
92
93      def backward_propagation(self, params, cache, y):
94          """
95          Parameters:
96          -----------
97          params : Dict[class[Parameters]]
98              params[l].w = Weights
99              params[l].bias = Boolean
100             params[l].b = Bias
101         cache : Dict[array_like]
102             cache['a'] : array_like
103             cache['dg'] : array_like
104         y : array_like
105
106         Returns:
107         --------
108         None
109         """
110
111         # Retrieve cache
112         a = cache['a']
113         dg = cache['dg']
114
115         # Initialize differentials along the network
116         delta = {}
117         delta[self.L] = (a[self.L] - y) / y.shape[1]
118
119         for l in reversed(range(1, self.L + 1)):
120             delta[l - 1] = dg[l- 1] * params[l].backward(delta[l], a[l - 1])
121
122     def update_parameters(self, params, learning_rate=0.1):
123         """
124         Parameters:
125         -----------
126         params : Dict[class[Parameters]]
127             params[l].w = Weights
128             params[l].bias = Boolean
129             params[l].b = Bias
130         learning_rate : float
131             Default : 0.01
132
133         Returns:
134         --------
```

```
        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.bi

        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]
        x : array_like
```

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

3.2 Implementation in Python via `tensorflow`

We implement a neural network using `tensorflow.keras`.

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

```
    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
    )

    ## 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

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

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:

```
## Shuffle, split and normalize data
class ProcessData():
    def __init__(self, x, y, test_percent, *dev_percents, seed=1, shuffle=True, feat
        """
        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 = list(dev_percents)
        self.k_fold = len(self.dev_percent)
        self.seed = seed
        self.shuffle = shuffle
        self.feat_as_col = feat_as_col

        self.split()
        self.normalize()

        print(f"x_train.shape: {self.train['x'].shape}")
        print(f"y_train.shape: {self.train['y'].shape}")
        print(f"x_test.shape: {self.test['x'].shape}")
        print(f"y_test.shape: {self.test['y'].shape}")
        for k in range(self.k_fold):
            print(f"x_dev[{k}].shape: {self.dev['x'][k].shape}")
            print(f"y_dev[{k}].shape: {self.dev['y'][k].shape}")

    def split(self):
```

```
        """
        Parameters:
        -----------
        None

        Returns:
        --------
        None
        """
        x_aux, x_test, y_aux, y_test = train_test_split(self.x, self.y, test_size=se
        left_over = 1 - self.test_percent
        x_dev = []
        y_dev = []
        for perc in self.dev_percent:
            aux_perc = perc / left_over
            x_aux, x_d, y_aux, y_d = train_test_split(x_aux, y_aux, test_size=aux_pe
            x_dev.append(x_d)
            y_dev.append(y_d)
            left_over -= perc

        if self.feat_as_col:
            self.train = {'x' : x_aux, 'y' : y_aux}
            self.test = {'x' : x_test, 'y' : y_test}
            self.dev = {'x' : x_dev, 'y' : y_dev}
        else:
            self.train = {'x' : x_aux.T, 'y' : y_aux.T}
            self.test = {'x' : x_test.T, 'y' : y_test.T}
            x_dev = [cv.T for cv in x_dev]
            y_dev = [cv.T for cv in y_dev]
            self.dev = {'x' : x_dev, 'y' : y_dev}

    def normalize(self, z=None, eps=0.0):
        """
        Parameters:
        -----------
        z : array_like
            Default : None - For initialization
        eps : float
            Default 0.0 - For stability

        Returns:
        z_scale : array_like
        """
        if z == None:
            x = self.train['x']
            axis = 0 if self.feat_as_col else 1
            self.mu = np.mean(x, axis=axis, keepdims=True)
```

```
92              self.var = np.var(x, axis=axis, keepdims=True)
93              self.theta = 1 / np.sqrt(self.var + eps)
94              self.train['x'] = self.theta * (x - self.mu)
95              self.test['x'] = self.theta * (self.test['x'] - self.mu)
96              for k in range(self.k_fold):
97                  self.dev['x'][k] = self.theta * (self.dev['x'][k] - self.mu)
98
99          else:
100             z_scale = self.theta * (z - self.mu)
101             return z_scale
```

5 Regularization

Suppose we're training an L-layer neural network with dataset $\{(x_j, y_j)\} \subset \mathbb{R}^{m_0} \times \mathbb{R}^{m_L}$ with N examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \to \mathbb{R}$, then we have our cost function $\mathbb{J}$ defined on our one-parameter families of parameters W and b given by

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

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

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

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

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

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 usage of regularization would be similar to the following outline:

i. Partition our dataset $\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}$.

ii. Give a set Λ of potential regularization parameters.

iii. For each $\lambda \in \Lambda$, we first train on $\mathbb{X}$, that is, we obtain

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

which dependent on λ.

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

v. After finding $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$ for each $\lambda \in \Lambda$, we choose our desired λ and hence our desired parameters W and b.

vi. We evaluate our model on $\mathcal{T}$ to determine the overall accuracy.

5.1 Python Implementation

```
import numpy as np

import mlLib.utils as utils
import mlLib.npActivators as npActivators

def forward_propagation(x, params, activators):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params['w'][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    Returns
    -------
    cache : Dict[Dict]
```

```
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params['w']
    b = params['b']
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Initialize a
    a[0] = x
    for l in range(1, L + 1):
        z[l], a[l] = utils.linear_activation_forward(a[l - 1], w[l], b[l], activator

    cache = {'a' : a, 'z' : z}
    return cache

def compute_cost(y, params, cache, lambda_=0.0):
    """
    Parameters
    ----------
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
    lambda_ : float
        Default: 0.0

    Returns
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
```

```
    a = cache['a']
    w = params['w']
    L = len(a)
    aL = a[L - 1]

    ## Regularization term
    R = 0
    for l in range(1, L):
        R += np.sum(w[l] * w[l])
    R *= (lambda_ / (2 * n))

    ## Unregularized cost
    J = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))

    ## Total Cost
    cost = J + R
    cost = float(np.squeeze(cost))
    return cost

def backward_propagation(x, y, params, cache, activators, lambda_=0.0):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    lambda_ : float
        Default: 0.0

    Returns
    -------
    grads : Dict[Dict]
        grads['dw'][l] : array_like
            dw[l].shape = w[l].shape
```

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

```
        Default: 0.0
    num_iters : int
        Number of iterations with which our model performs gradient descent
        Default: 10000
    print_cost : Boolean
        If True, print the cost every 1000 iterations
        Default: False

    Returns
    -------
    params : Dict[Dict]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    """
    ## Set dimensions and Initialize parameters
    n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
    params = utils.initialize_parameters_random(layers)

    # main gradient descent loop
    for i in range(num_iters):
        cache = forward_propagation(x, params, activators)
        cost = compute_cost(y, params, cache, lambda_)
        grads = backward_propagation(x, y, params, cache, activators, lambda_)
        params = utils.update_parameters(params, grads)

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

    return params, cost
```

5.2 (Inverted) Dropout Regularization

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

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

Let Q_0, Q_1, Q_2 denote the collection of all nodes in Layers $0, 1, 2$, respectively. Let $p_0, p_1, p_2 \in [0, 1]$, and define a probability distribution $\mathbb{P}_\ell$ 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 layer ℓ and each training example x_j define the "dropout vector" ${D^{[\ell]}}_j$ by

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

where

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

ii. During forward propagation, we redefine

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

iii. During backward propagation, we define

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

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

5.2.1 Python Implementation

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

```
import numpy as np

import mlLib.utils as utils

def dropout_matrices(layers, num_examples, keep_prob):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = number of nodes in layer l
    num_examples : int
        The number of training examples
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    D : Dict[array_like]
        D[l].shape = (layers[l], num_ex)
        D[l] = a Boolean array
    """
    np.random.seed(1)
    L = len(layers)
    D = {}
    for l in range(L - 1):
        D[l] = np.random.rand(layers[l], num_examples)
        D[l] = (D[l] < keep_prob[l]).astype(int)
        assert(D[l].shape == (layers[l], num_examples))
    return D



def forward_propagation(x, params, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params['w'][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer_dims[l], num_ex)
```

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

```
    params : Dict
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer[l], num_ex)
        D[l] = a Boolean array
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    grads : Dict[Dict]
        grads['dw'][l] : array_like
            dw[l].shape = w[l].shape
        grads['db'][l] : array_like
            db[l].shape = b[l].shape
    """
    ## Retrieve parameters
    a = cache['a']
    z = cache['z']
    w = params['w']
    n = x.shape[1]
    L = len(z)

    ## Compute deltas
    delta = {}
    delta[L] = a[L] - y
    for l in reversed(range(1, L)):
        deltal = utils.linear_activation_backward(delta[l + 1], z[l], w[l + 1], acti
        deltal = deltal * D[l]
        deltal /= keep_prob[l]
        delta[l] = deltal

    ## Compute gradients
    dw = {}
    db = {}

    for l in range(1, L + 1):
```

```
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * delta[l] @ a[l - 1].T
        assert(dw[l].shape == w[l].shape)
    grads = {'dw' : dw, 'db' : db}
    return grads

def model(x, y,
            hidden_sizes,
            activators,
            keep_prob = 1.0,
            num_iters=2500,
            learning_rate=0.1,
            print_cost=False):
    """
    Parameters
    ----------
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_sizes : List[int]
        The number nodes layer l = hidden_sizes[l-1]
    activators : List[function]
        activators[l] = activation function of layer l+1
    keep_prob : List[float] | float
        keep_prob[l] = The probabilty of keeping a node in layer l
        keep_prob = The same probability for all input and hidden layers
    num_iters : int
        Number of iterations with which our model performs gradient descent
    learning_rate : float
        The learning rate for gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations

    Returns
    -------
    params : Dict[Dict]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    cost : float
        The final cost value for the optimized parameters returned
    """
```

```
    ## Retrieve parameters
    n, layers = utils.dim_retrieval(x, y, hidden_sizes)
    params = utils.initialize_parameters_random(layers)

    ## Expand keep_prob to a list if it's a single float
    if isinstance(keep_prob, float):
        keep_prob = [keep_prob] * (len(layers) - 1)
    ## Main gradient descent loop
    for i in range(num_iters):
        D = dropout_matrices(layers, n, keep_prob)
        cache = forward_propagation(x, params, activators, D, keep_prob)
        cost = utils.compute_cost(y, cache)
        grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
        params = utils.update_parameters(params, grads, learning_rate)

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

    return params, cost
```

5.3 Data Augmentation

This section requires work.

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

5.4 Early Stopping

This section requires work.

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

6 Gradients and Numerical Remarks

This section requires work. See "He Initialization" and "Xavier Initialization"

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

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

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

6.1 Numerical Gradient Checking

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ 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

```
## f(x) = x_1*x_2*...*x_n
def fctn(x):
    n = x.shape[0]
    y = np.prod(x)
    grad = np.zeros((n, 1))
    for i in range(n):
        omit = 1 - np.eye(1, n, i).T
        omit = np.array(omit, dtype=bool)
        grad[i, 0] = np.prod(x, where=omit)
    return y, grad

def gradient_check(grad, f, x, epsilon=1e-3):
    """
    Parameters
    ----------
    grad : array_like
        grad.shape= (n, 1)
    f : function
        The function to check.
    x : array_like
        x.shape = (n, 1)
    epsilon : float
        Default 0.001
    Returns
    error : float
    -------
    """
    n = x.shape[0]
    y_diffs = []
    for i in range(n):
        e = np.eye(1, n, i).T
        x_plus = x + epsilon * e
        x_minus = x - epsilon * e
        y_plus, _ = f(x_plus)
        y_minus, _ = f(x_minus)
        y_diffs.append(y_plus - y_minus)
    y_diffs = np.array(y_diffs).reshape(n, 1)
    y_diffs = y_diffs / (2 * epsilon)

    error = (np.linalg.norm(y_diffs - grad)
                / (np.linalg.norm(y_diffs) + np.linalg.norm(grad)))
    return error
```

7 Gradient Descent

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

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

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

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

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

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

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

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

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.

7.1 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

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

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

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

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 $t := i + k$. That is, we have the following algorithm:

1. Initialize our parameters $W^{\{0\}}$ and $b^{\{0\}}$. Initialize ${V_W}^{\{0\}} = {V_b}^{\{0\}} = 0$. Fix a momentum hyper-parameter $\beta_1 \in [0, 1)$.

2. For $0 \leq i <$ `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

$$W^{\{t\}} = W^{\{t-1\}} - \alpha {V_W}^{\{t\}}$$
$$b^{\{t\}} = b^{\{t-1\}} - \alpha {V_b}^{\{t\}}$$

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

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

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

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

where where taking the Hadamard-root. Note that this root is necessary for our update to make sense (consider the units involved in such an equation), but it does introduce the potential to divide by zero (which we'll fix by a small . 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

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

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

7.4 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

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

      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

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

7.5 Learning Rate Decay

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

- $$\alpha(i) = \frac{1}{1 + \eta i}\alpha_0,$$
- $$\alpha(i) = \alpha_0 \eta^i,$$
- $$\alpha(i) = \frac{\eta}{\sqrt{i}}\alpha_0.$$

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

7.6 Python Implementation

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

```
import mlLib.utils as utils

def get_batches(x, y, b):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, n)
    y : array_like
        y.shape = (k, n)
    b : int

    Returns
    -------
    batches : List[Dict]
        batches[i]['x'] : array_like
            x.shape = (m, b) # except last batch
            y.shape = (k, b) # except last batch

    """
    m, n = x.shape
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
    x = x.T
    y = y.T

    B = int(np.ceil(n / b))
    batches = []
    for i in range(B):
        x_temp = x[:,(b * i):(b * (i + 1))]
        y_temp = y[:,(b * i):(b * (i + 1))]
        batches.append({'x' : x_temp, 'y' : y_temp})
    # Slicing automatically ends at the end of
    # the list if the stop is outside the index
    return batches

def initialize_momenta(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    v : Dict[Dict[array_like]]
    s : Dict[Dict[array_like]]
    """
```

```
    vw = {}
    vb = {}
    sw = {}
    sb = {}
    for l in range(1, len(layers)):
        vw[l] = np.zeros((layers[l], layers[l - 1]))
        sw[l] = np.zeros((layers[l], layers[l - 1]))
        vb[l] = np.zeros((layers[l], 1))
        sb[l] = np.zeros((layers[l], 1))

    v = {'w' : vw, 'b' : vb}
    s = {'w' : sw, 'b' : sb}

    return v, s

def learning_rate_decay(epoch, learning_rate=0.01, decay_rate=0.0):
    """
    Parameters
    ----------
    eposh : int
    learning_rate : float
        Default: 0.01
    decay_rate : float
        Default: 0.0 - Returns a constant learning_rate

    Returns
    -------
    learning_rate : float
    """
    learning_rate = (1 / (1 + epoch * decay_rate)) * learning_rate
    return learning_rate

def corrected_momentum(v, grads, update_iter, beta1=0.0):
    """
    Parameters
    ----------
    v : Dict[Dict[array_like]]
        v['w'][l].shape = w[l].shape
        v['b'][l].shape = b[l].shape
    grads : Dict[Dict]
        grads['w'][l] : array_like
            dw[l].shape = w[l].shape
        grads['b'][l] : array_like
            db[l].shape = b[l].shape
    update_iter : int
    beta1 : float
        Default: 0.0 - Returns grads
```

```
        Usual: 0.9

    Returns
    -------
    v : Dict[Dict[array_like]]
        v['w'][l].shape = dw[l].shape
        v['b'][l].shape = db[l].shape
    """
    ## Retrieve velocities and gradients
    vw = v['w']
    vb = v['b']
    dw = grads['w']
    db = grads['b']
    L = len(dw)

    for l in range(1, L + 1):
        vw[l] = beta1 * vw[l] + (1 - beta1) * dw[l]
        vw[l] /= (1 - beta1 ** update_iter)
        assert(vw[l].shape == dw[l].shape)
        vb[l] = beta1 * vb[l] + (1 - beta1) * db[l]
        vb[l] /= (1 - beta1 ** update_iter)
        assert(vb[l].shape == db[l].shape)

    v = {'w' : vw, 'b' : vb}
    return v

def corrected_rmsprop(s, grads, update_iter, beta2=0.999):
    """
    Parameters
    ----------
    s : Dict[Dict[array_like]]
        s['w'][l].shape = w[l].shape
        s['b'][l].shape = b[l].shape
    grads : Dict[Dict]
        grads['w'][l] : array_like
            dw[l].shape = w[l].shape
        grads['b'][l] : array_like
            db[l].shape = b[l].shape
    update_iter : int
    beta2 : float
        Default: 0.999

    Returns
    -------
    s : Dict[Dict[array_like]]
        s['w'][l].shape = w[l].shape
        s['b'][l].shape = b[l].shape
```

```
147     """
148     ## Retrieve accelerations and gradients
149     sw = s['w']
150     sb = s['b']
151     dw = grads['w']
152     db = grads['b']
153     L = len(dw)
154 
155     for l in range(1, L + 1):
156         sw[l] = beta2 * sw[l] + (1 - beta2) * (dw[l] * dw[l])
157         sw[l] /= (1 - beta2 ** update_iter)
158         assert(sw[l].shape == dw[l].shape)
159         sb[l] = beta2 * sb[l] + (1 - beta2) * (db[l] * db[l])
160         sb[l] /= (1 - beta2 ** update_iter)
161         assert(sb[l].shape == db[l].shape)
162 
163     s = {'w' : sw, 'b' : sb}
164     return s
165 
166 
167 def update_parameters_adam(params, grads, epoch, batch_iter, v, s, momenta=[1e-8, 0.
168     """
169     Parameters
170     ----------
171     params : Dict[Dict]
172         params['w'][l] : array_like
173             w[l].shape = (layers[l], layers[l-1])
174         params['b'][l] : array_like
175             b[l].shape = (layers[l], 1)
176     grads : Dict[Dict]
177         grads['dw'][l] : array_like
178             dw[l].shape = w[l].shape
179         grads['db'][l] : array_like
180             db[l].shape = b[l].shape
181     epoch : int
182     batch_iter : int
183     learning_rate : float
184         Default: 0.01
185     momenta : List[float]
186         momenta[0] = epsilon
187             Default: 10^{-8}
188         momenta[1] = beta_1
189             Default: 0.9
190         momenta[2] = beta_2
191             Default: 0.999
192 
193     Returns
```

```
    -------
    params : Dict[Dict]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    """
    update_iter = epoch + batch_iter
    ## Retrieve parameters
    w = copy.deepcopy(params['w'])
    b = copy.deepcopy(params['b'])
    L = len(w)

    ## Update velocites and accelerations
    v = corrected_momentum(v, grads, update_iter, momenta[1])
    vw = v['w']
    vb = v['b']
    s = corrected_rmsprop(s, grads, update_iter, momenta[2])
    sw = s['w']
    sb = s['b']

    ## Update learning rate
    learning_rate = learning_rate_decay(epoch, alpha0, decay_rate)

    ## Perform update
    for l in range(1, L + 1):
        w[l] = w[l] - learning_rate * vw[l] / (np.sqrt(sw[l]) + momenta[0])
        b[l] = b[l] - learning_rate * vb[l] / (np.sqrt(sb[l]) + momenta[0])

    params = {'w' : w, 'b' : b}
    return params

def model(x, y,
        hidden_layer_sizes,
        activators,
        batch_size,
        lambda_=0.0,
        num_iters=10000,
        print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
```

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

```
                                        momenta=[1e-8, 0.9, 0.999],
                                        learning_rate=0.01,
                                        decay_rate = 0.0)
            batch_iter += 1

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

    return params, cost
```

8 Tuning Hyper-Parameters

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

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

Furthermore, suppose we impose a neural network architecture which has a collection of hyper-parameters (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.

8.1 Python Implementation

```
def hyperparameter_scale(k, p):
    """
    Parameters
    ----------
    k : int
        The number random points to generate
    p : int
        The smallest magnitude for our log-scale

    Returns
    -------
    hypers : List[float]
        The list of hyper-parameters with which to tune
    """
    hypers = []
    for _ in range(k):
        r = p * np.random.rand()
        hypers.append(10 ** r)
    return hypers
```

9 Batch Normalization

See [1].

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

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

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

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

we can see fairly trivially that we can recover x_j (thus allowing for identity activation units), indeed, let $\gamma = \sigma$ and $\beta = \mu$, and hence

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

as desired. Moreover, we see that we can actually control what mean and variance we wish to impose on our input-vectors x. Indeed, let $\hat{x}$ denote the

normalized x, and consider

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

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

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

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

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

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

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

The procedure for forward propagation should be immediately obvious from the closer look at layer-ℓ. However, we notice that

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

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

9.1 Backward Propagation

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

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

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

where

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

and

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

Define the compositional function

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

given by

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

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

This leads to compute some auxiliary differentials before continuing further.

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

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

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

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

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

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

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

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

Then the differentials

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

are given by

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

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

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

For fixed z*, define the variance*

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

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

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

Moreover, for $\alpha = 1, ..., N$*, the differentials*

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

are given by

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

Proof: Immediate from direct calculation. □

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

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

with

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

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

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

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

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

are given by the diagonal matrices

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

Proof: We compute directly after noting that

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

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

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

as desired. □

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

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

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

$$BN^{\gamma,\beta}_\alpha : \mathbb{R}^{m \times N} \to \mathbb{R}^m$$

denote

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof: Follows immediately from the previous Corollary. □

We now return to considering the compositional function

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

given by

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

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

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

  and hence

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

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

  yielding

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

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

  and so

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

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

  thus

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

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

  and hence

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

Finally, since

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

we've described our desired gradients after summation.

9.2 Inferencing

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

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

and the variances

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

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

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

from which we average again to obtain

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

and the collection

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

from which we use the unbiased estimate

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

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

9.3 Algorithm Outline

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

1. Set hyper-parameters. Initialize parameters.
2. For $0 \leq i \leq$ `num_iters`:
   a. Generate batches $\{\mathbb{X}^k : 1 \leq k \leq K\}$.
   b. For $1 \leq k \leq K$:
      i. Perform forward propagation on $\mathbb{X}^k$:
         - $$z^{[1]} = W^{[1]} x$$
         - For $\ell \in \{1, ..., L-1\}$:
           – $$z^{[\ell]} = W^{[\ell]} a^{[\ell-1]}$$

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

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

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

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

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

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

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

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

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

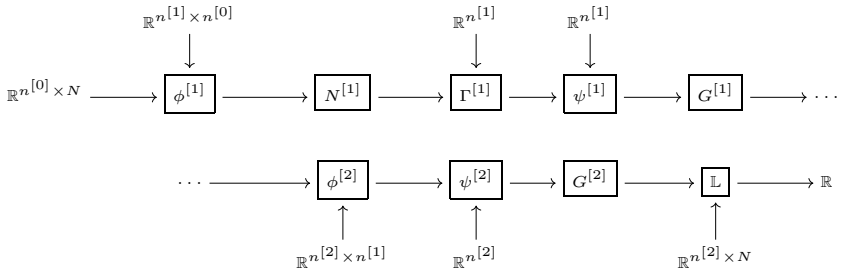
iv. Update parameters.

3. Compute
$$\mu^{[\ell]} = \mathbb{E}[\mu^{[\ell]}{}_k],$$
$$\sigma^{2[\ell]} = \frac{n}{n-1}\mathbb{E}[\sigma^{2[\ell]}{}_k]$$

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

9.4 Better Backpropagation

We consider a neural network utilizing batch normalization of the form



where we have the functions

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

   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

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

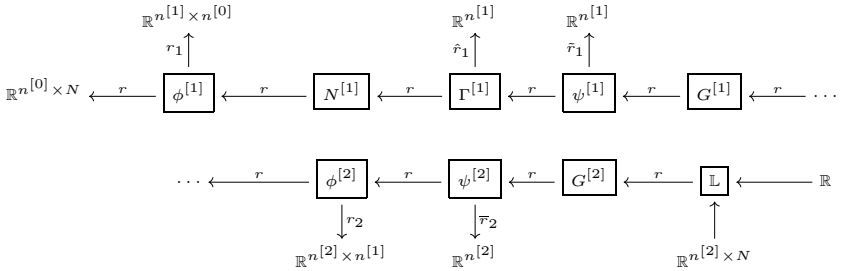
   where

   $$\vec{1}^T = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix} \in \mathbb{R}^{n^{[\ell]}}.$$

5.
$$N^{[1]} : \mathbb{R}^{n^{[1]} \times N} \to \mathbb{R}^{n^{[1]} \times N}$$
is the normalization operator given by
$$N^{[1]} : x^i_j \mapsto \frac{x^i_j - \mathbb{E}[x^i]}{\sqrt{\mathbb{V}[x^i] + \epsilon}},$$
where $\mathbb{E}$ is the expectation operator, i.e.,
$$\mathbb{E}[x^i] = \frac{1}{N} \sum_{j=1}^{N} x^i_j,$$
and $\mathbb{V}$ is the variance operator, i.e.,
$$\mathbb{V}[x^i] = \mathbb{E}[(x^i - \mathbb{E}[x^i])^2].$$

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
$$\vec{1}^T = \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \in \mathbb{R}^{n^{[\ell]}}.$$

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 $Z \in T_W\mathbb{R}^{m\times n}$, we have that

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

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

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

3. Suppose $\psi : \mathbb{R}^n \times \mathbb{R}^{n\times N} \to \mathbb{R}^{n\times N}$ 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 $((\gamma, x), (\eta, \xi)) \in T(\mathbb{R}^n \times \mathbb{R}^{n\times N})$ and $\zeta \in T_{\Gamma(\gamma,x)}\mathbb{R}^{n\times N}$ that:

(a) In the network direction, we have that

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

with reverse differential

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

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

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

with reverse differential

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

5. As the normalization operator is quite involved, we move its computation to the appendix, **??**.

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

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

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

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

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

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

is given by

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

and we compute the reverse differentials for a learning rate $\alpha \in T_{\mathbb{J}}\mathbb{R}$ with the assumption that our second activator function is the sigmoid function. Indeed,

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

This leads us to

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

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

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

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

and finally,

$$
\begin{aligned}
r_1 \mathbb{J}_{W^{[1]}}(\alpha) &= \frac{\alpha}{N} r_1 \phi^{[1]}_{(W^{[1]},x)} \circ r N^{[1]}_{u^{[1]}} \circ r\Gamma^{[1]}_{(\gamma^{[1]},z^{[1]})} \left(G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} r_1 \phi^{[1]}_{(W^{[1]},x)} \circ r N^{[1]}_{u^{[1]}} \left(\gamma \vec{1}^T \odot G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} R_{x^T} \circ r N^{[1]}_{u^{[1]}} \left(\gamma \vec{1}^T \odot G^{[1]\prime}(\tilde{z}^{[1]}) \odot W^{[2]T}(a^{[2]} - y) \right) \\
&= \frac{\alpha}{N} \sum_{j,l=1}^{N} \sum_{i=1}^{n^{[1]}} T_i^{jk}{}_l \gamma^i g^{[1]\prime}(\tilde{z}^{[1]i}{}_j) W^{[2]}{}_i (a^{[2]}{}_j - y_j) x_l^m
\end{aligned}
$$

9.5 Python Implementation

Work in Progress

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

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

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.

References

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