Neural Networks

Matt R

July 8, 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

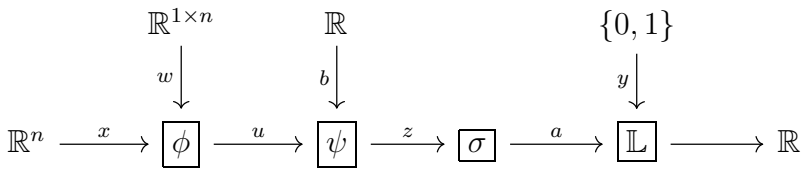
We wish to compute the gradient of our cost function $\mathbb{J}$ with respect to our trainable parameters, $w \in \mathbb{R}^{1\times n}$ and $b \in \mathbb{R}$. To this end, we define the functions

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

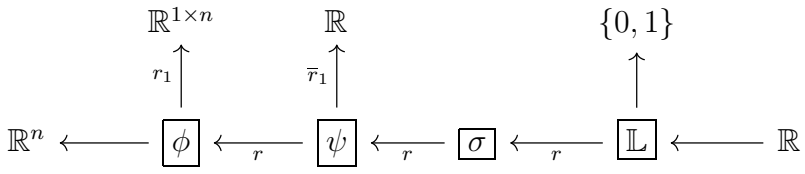
and

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

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:
```

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

1.3 Implementation in Python via `sklearn`

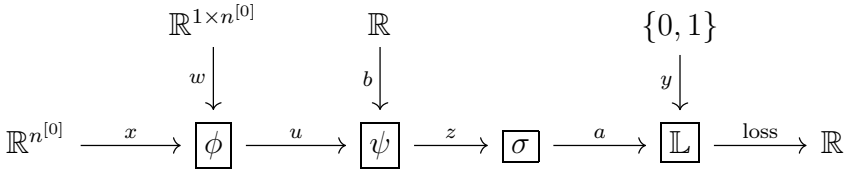
Here we include the general method of coding a logistic regression model via `scikit-learn`'s modeling library.

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

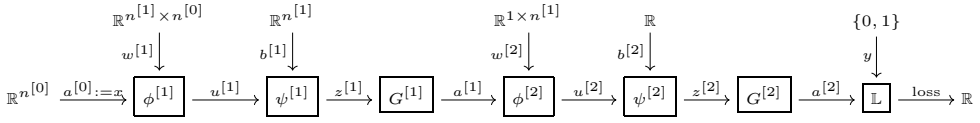
```
    x = dataset[:, :10]
    y = dataset[:, 10]

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

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

2 Neural Networks: A Single Hidden Layer

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



Such a structure will be called a *network*, and the a is known as the *activation node*. Logistic regression can be too simplistic of a model for many situations, e.g., if the dataset isn't linearly separable (i.e., there doesn't exist some well-defined decision boundary built from a linear-surface), then logistic regression won't give a high-accuracy model. To modify this model to handle more complex situations, we introduce a new "hidden layer" of nodes with their own (possibly different) activation functions. That is, we consider a network of the following form:



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

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

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

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

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

where $v \in \mathbb{R}^n$ and $\{e_i : 1 \leq i \leq n\}$ is the standard basis for $\mathbb{R}^n$. In practice, we will sometimes write $g = G$ for a broadcasted function, and let the context determine the meaning of g.

castingDifferential

Lemma 2.2. *Suppose $g : \mathbb{R} \to \mathbb{R}$ is any smooth function and $G : \mathbb{R}^n \to \mathbb{R}^n$ is the broadcasting of g from $\mathbb{R}$ to $\mathbb{R}^n$. Then the differential $dG_z : T_z\mathbb{R}^n \to T_{G(z)}\mathbb{R}^n$ is given by*

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

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

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

We use the notation

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

and thus may write

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

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

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

Proof: We calculate

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

and letting $e_1, ...e_m$ denote the usual basis for $T_z\mathbb{R}^m$ (identified with $\mathbb{R}^m$), we see that

$$\begin{aligned} dG_z(e_j) &= [g'(z^i)] \odot e_j \\ &= g'(z^j)e_j, \end{aligned}$$

from which conclude that dG_z is diagonal with (j,j)-th entry $g'(z^j)$ as desired.

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

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

and the result follows. □

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

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

2.1 Activation Functions

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

2.1.1 The Sigmoid Function

We have the sigmoid function $\sigma(z)$ given by

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

We note that since

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

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

2.1.2 The Hyperbolic Tangent Function

We have the hyperbolic tangent function $\tanh(z)$ given by

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

We then calculate

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

Furthermore, we note that

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

Indeed,

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

as desired.

2.1.3 The Rectified Linear Unit Function

We have the leaky-ReLU function $\mathrm{ReLU}(z; \beta)$ given by

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

for some $\beta > 0$ (typically chosen very small).

We have the rectified linear unit function $\text{ReLU}(z)$ given by setting $\beta = 0$ in the leaky-ReLu function, i.e.,

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

We then calculate

$$\begin{aligned} \text{ReLU}'(z; \beta) &= \begin{cases} \beta & z < 0 \\ 1 & z \geq 0 \end{cases} \\ &= \beta \chi_{(-\infty, 0)}(z) + \chi_{[0, \infty)}(z), \end{aligned}$$

where

$$\chi_A(z) = \begin{cases} 1 & z \in A \\ 0 & z \notin A \end{cases},$$

is the indicator function.

2.1.4 The Softmax Function

We finally have the softmax function $\text{softmax}(z)$ given by

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

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

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

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

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

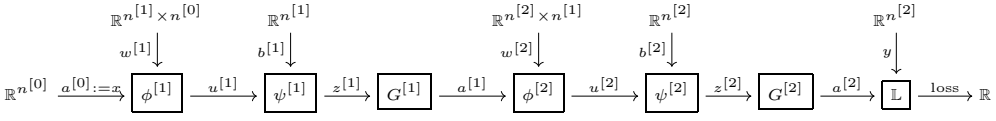
That is, as a map $dS_z : T_z\mathbb{R}^m \to T_{S(z)}\mathbb{R}^m$, we have that

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

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

2.2 Backward Propagation

We consider a neural network of the form



where we have the functions:

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

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

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

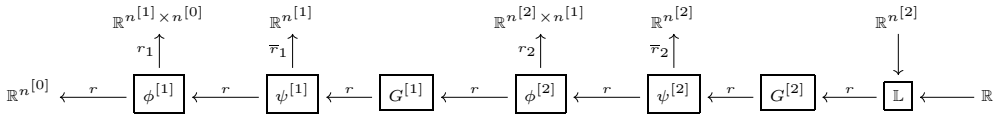
   is given by

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

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

4.
$$\mathbb{L} : \mathbb{R}^{n^{[2]}} \times \mathbb{R}^{n^{[2]}} \to \mathbb{R}$$
is the given loss-function.

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



First, we need to consider our individual derivatives:

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

2. Suppose $\phi : \mathbb{R}^{m \times n} \times \mathbb{R}^n \to \mathbb{R}^m$ is given by
$$\phi(w, x) = wx.$$
Then we have two differentials to consider:

   (a) For any $(w, x) \in \mathbb{R}^{m \times n} \times \mathbb{R}^n$ and any $\xi \in T_x\mathbb{R}^n$, we have that
$$\begin{aligned} d\phi_{(w,x)}(\xi) &= w\xi \\ &= L_w(\xi); \end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and we have our cost function

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

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

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

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

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

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

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

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

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

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

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

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. $r_2\mathbb{J}_p$:

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

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

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

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. $r_1\mathbb{J}_p$:

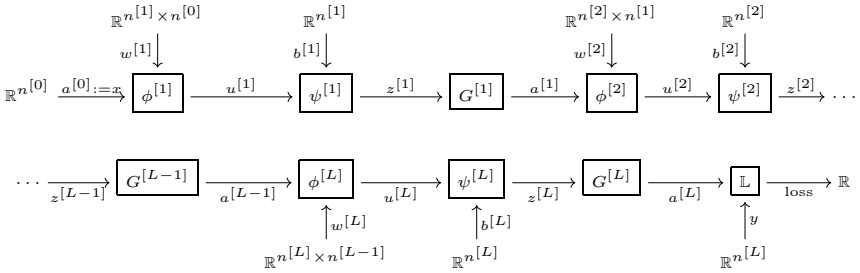
$$\begin{aligned} r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1\phi^{[1]}_p \circ r\psi^{[1]}_{u^{[1]}} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{a^{[1]}} \circ r\psi^{[2]}_{u^{[2]}} \circ rG^{[2]}_{z^{[2]}} \circ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

and hence

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

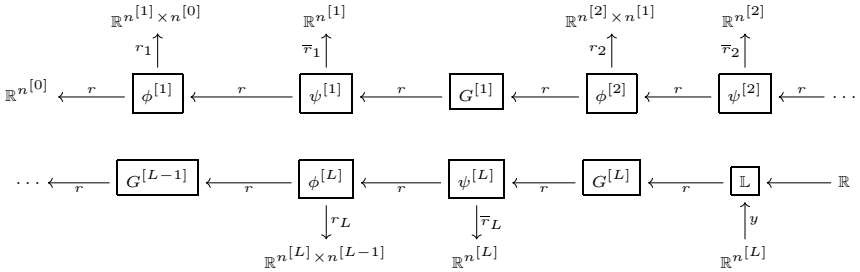
3 Deep Neural Networks

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

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

2. Compute

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

and

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

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

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

4. Compute

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

and

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

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

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

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

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

If however, we have a very large dataset (i.e., $N > 10^4$), then we assume a much smaller ratio of something similar to

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

In general, we use our training set $\mathbb{X}$ to train our parameters $w^{[\ell]}$ and $b^{[\ell]}$, we use our development set $\mathcal{D}$ to tune our hyper-parameters (i.e., learning rate, number of layers, number of nodes per layer, activation function, number of iterations to perform gradient descent, regularization parameters, etc), and we use our test set $\mathcal{T}$ to evaluate the accuracy of our model. Since we're partitioning our dataset to better increase the accuracy of our model, we need to define an error function. To this end, define $\mathcal{E} : 2^{\mathbb{D}} \to [0, 1]$ by

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

where $\varepsilon : \mathbb{D} \to \{0, 1\}$ is defined by

$$\varepsilon(x, y) = \begin{cases} 1 & \text{if } y = \hat{y}(x) \\ 0 & \text{else.} \end{cases}$$

From our partition and error function we can make several claims of the fitting of our model to our data. Indeed, let $\epsilon > 0$ be a small percentage (with exact value depending on specific examples), then:

- If $\mathcal{E}(\mathbb{X}) < \epsilon$ and $\mathcal{E}(\mathbb{X}) < \mathcal{E}(\mathcal{D}) \lessapprox 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.
- If $\mathcal{E}(\mathbb{X}) \approx \mathcal{E}(\mathcal{D}) \gtrapprox 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.
- If $10\epsilon \lessapprox \mathcal{E}(\mathbb{X}) \ll \mathcal{E}(\mathcal{D})$, then we say our model has both high bias (since it doesn't fit our training data well) and high variance (because the model fits the training data better than the development data).
- If $\mathcal{E}(\mathbb{X}), \mathcal{E}(\mathcal{D}) < \epsilon$, then we say the model has both low bias and low variance.

Remark 4.1. *The interpretations of our error percentage is based on two crucial assumptions:*

- *$\mathcal{D}$ and $\mathcal{T}$ come from samplings with the same distribution of outputs (i.e., if we're determining whether a collection of images contain a cat, we should never have that $\mathcal{D}$ is mostly cat pictures, and $\mathcal{T}$ is mostly non-cat pictures).*
- *The optimal error for the model is approximately 0%. That is, if a human were looking at the data, they could determine the correct response with negligible error. This is sometimes called the Bayes error.*

If either of these assumptions fail to hold, other methods of analysis may be required to obtain meaningful insights for the performance of our model.

A methodology for using errors could be as follows

1. Check $\mathcal{E}(\mathbb{X})$ for high bias.
   a. If "Yes", then we can try a bigger network, we can train longer, or we can change the neural network architecture. Then we return to (1.).
   b. If "No", then we move to (2.).
2. Check $\mathcal{E}(\mathcal{D})$ for high variance.
   a. If "Yes", then we can try to get more data, try regularization, or try changing the neural network architecture. Then we return to (1.).
   b. If "No", then we're done.

4.1 Python Implementation

To implement a partitioning we could do something like the following:

```
## Classes

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

        Returns:
        --------
        None
        """
        self.x = x
        self.y = y
        self.test_percent = test_percent
        self.dev_percent = dev_percent
        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}")
        if self.dev_percent > 0.0:
            print(f"x_dev.shape: {self.dev['x'].shape}")
            print(f"y_dev.shape: {self.dev['y'].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
        aux_perc = self.dev_percent / left_over
        x_train, x_dev, y_train, y_dev = train_test_split(x_aux, y_aux, test_size=au

        if self.feat_as_col:
            self.train = {'x' : x_train, 'y' : y_train}
            self.test = {'x' : x_test, 'y' : y_test}
            self.dev = {'x' : x_dev, 'y' : y_dev}
        else:
            self.train = {'x' : x_train.T, 'y' : y_train.T}
            self.test = {'x' : x_test.T, 'y' : y_test.T}
            self.dev = {'x' : x_dev.T, 'y' : y_dev.T}

    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)
            self.var = np.var(x, axis=axis, keepdims=True)
            self.theta = 1 / np.sqrt(self.var + eps)
            self.train['x'] = self.theta * (x - self.mu)
            self.test['x'] = self.theta * (self.test['x'] - self.mu)
            self.dev['x'] = self.theta * (self.dev['x'] - self.mu)

        else:
```

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

5 Regularization

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

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

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

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

for some $\lambda > 0$, where $\|\cdot\|_F$ represents the Frobenius norm on matrices, and we define the *regularized cost function* $\mathbb{J}^R$ given by

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

Adding such an $R(w)$ to our cost function is known as L^2*-regularization*. We note that by linearity we have the following equalities amongst gradients:

$$\frac{\partial \mathbb{J}^R}{\partial b^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial b^{[\ell]}}$$

and

$$\frac{\partial \mathbb{J}^R}{\partial w^{[\ell]}} = \frac{\partial \mathbb{J}}{\partial w^{[\ell]}} + \frac{\lambda}{N} w^{[\ell]}.$$

The idea behind regularization is that we're now minimizing

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

and so for suitably chosen $\lambda > 0$, it forces $\left\| w^{[\ell]} \right\|_F$ to be small, along with minimizing the cost $\mathbb{J}$. This balancing-act of minimizing the two functions simultaneously helps with overfitting the data.

A typical tuning via regularization would be similar to the following outline:

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

ii. Give a set Λ of potential regularization parameters.

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

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

which is dependent on λ.

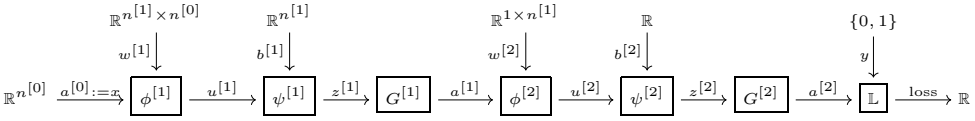
iv. Then using the aforementioned $(w, b) = (w, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.

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

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

5.1 (Inverted) Dropout Regularization

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



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

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

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

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

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

where

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

ii. During forward propagation, we redefine

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

iii. During backward propagation, we define

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

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

5.1.1 Python Implementation

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation


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

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

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

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

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

        return D

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

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

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

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

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

        return cost

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Data Augmentation

> This section requires work.

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

5.3 Early Stopping

This section requires work.

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

6 Gradients and Numerical Remarks

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

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

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

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

6.1 Numerical Gradient Checking

Suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a smooth function. Then, we recall the definition of the partial derivative

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

and so for sufficiently small $\epsilon > 0$, we have the approximation

$$\frac{\partial f}{\partial x^j} \approx \frac{f(x + \epsilon e_j) - f(x - \epsilon e_j)}{2\epsilon}.$$

Define the approximation function $F : \mathbb{R}^n \times (0, 1) \to \mathbb{R}^n$ by

$$F(x, \epsilon) = \frac{1}{2\epsilon} \begin{bmatrix} f(x + \epsilon e_1) - f(x - \epsilon e_1) \\ \vdots \\ f(x + \epsilon e_n) - f(x - \epsilon e_n) \end{bmatrix}.$$

Then we may check that our gradient computation $\nabla f(x)$ is correct by checking that

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

6.2 Python Implementation

Reformat this section for the reverse differential of varying sizes once the Reverse Differential section is completed. There should be several cases included:

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

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

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

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

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

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

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

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

iv. Perform gradient descent:

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

We make several remarks about mini-batch gradient descent:

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

  $$b = 2^p,$$

  for some $p \in \{6, 7, 8, 9, 10\}$ (usually $p < 10$).
- Choose a batch size b that ensures your computer's CPU/GPU can hold a dataset of that size.

7.0.1 Python Implementation via numpy

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation

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

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

        np.random.seed(seed)

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

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

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

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

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

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

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

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

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

        return cost

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

        Returns:
```

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

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

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

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

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

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

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

```

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

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

7.1 Weighted Averages

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

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

has variable T.

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

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

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

One way (and likely the most trivial) to achieve such a weighing is to have only the most recent k examples affect our model. That is, for fixed

$k \in \mathbb{N}$, and $t \geq k$, define the vector $\hat{x}_{t+1} \in \mathbb{R}^m$ by

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

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

The simple moving average satisfies our weight requirement of focusing more on the most recent data, however, older data, though being less relevant, should still affect our model, but in a reduced form. The simple model does not satisfy this more refined requirement. Let's modify the simple model as follows: Fix $\beta_1 \in [0, 1)$ and we initialize a $v_0 = 0 \in \mathbb{R}^m$, and define recursively the vector $v_t \in \mathbb{R}^m$ given by

$$v_t = \beta_1 v_{t-1} + (1 - \beta_1) x_t.$$

We claim that v_t can be interpreted as the next predicted value $\hat{x}_{t+1}$. Indeed, expanding our recursive definition

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

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

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

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

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

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

Remark 7.1. *We note that since we initialize our EMA with $v_0 = 0$, that our predictive model is very bad for small t. This usually is irrelevant for many models, but if we need to correct for bias, we may make the modification of*

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

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

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

and so

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

We then see that

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

which is the explicit definition of a weighted-average.

7.2 Gradient Descent with Momentum

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

We first recall our gradient descent algorithm:

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

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

1. Initialize our parameters $w^{\{0\}}$ and $b^{\{0\}}$. Initialize ${v_w}^{\{0\}} = {v_b}^{\{0\}} = 0$. Fix a momentum hyper-parameter $\beta_1 \in [0, 1)$.
2. For $0 \leq i <$ `num_iters`:
   a. Let $B = \left\lceil \frac{n}{b} \right\rceil$, and generate batches $\{\mathbb{X}^k\}$.
   b. For $1 \leq k \leq B$:

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

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

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

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

iv. Define

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

v. We update parameters

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

7.2.1 Python Implementation via `numpy`

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation

class Momentum():
    def __init__(self, params, bias, beta1=0.9):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w : array_like
            params[l].b : array_like
        bias : List[Boolean]
        beta1 : float
            Default: 0.9

        Returns:
        None
```

```
        """
        self.beta1 = beta1
        self.bias = bias
        self.w = {}
        self.b = {}
        for l, param in params.items():
            self.w[l] = np.zeros(param.w.shape)
            if self.bias[l]:
                self.b[l] = np.zeros(param.b.shape)

    def update(self, params, learning_rate=0.01):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].dw : array_like
            params[l].db : array_like
        learning_rate : float
            Default: 0.01

        Returns:
        None
        """
        for l, param in params.items():
            vw = self.beta1 * self.w[l] + (1 - self.beta1) * param.dw
            self.w[l] = vw
            w = param.w - learning_rate * self.w[l]
            param.w = w
            if self.bias[l]:
                vb = self.beta1 * self.b[l] + (1 - self.beta1) * param.db
                self.b[l] = vb
                b = param.b - learning_rate * self.b[l]
                param.b = b

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

```

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

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

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

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

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

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

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

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

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

```
162
163         # Compute unregularized cost
164         a = np.clip(a, eps, 1 - eps)    # Bound a for stability
165         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
166
167         cost = float(np.squeeze(J + R))
168
169         return cost
170
171     def backward_propagation(self, params, cache, y, dropout):
172         """
173         Parameters:
174         -----------
175         params : Dict[LinearParameters]
176             params[l].w = Weights
177             params[l].bias = Boolean
178             params[l].b = Bias
179         cache : Dict[array_like]
180             cache['a'] : array_like
181             cache['dg'] : array_like
182         y : array_like
183
184         Returns:
185         --------
186         None
187         """
188
189         # Retrieve cache
190         a = cache['a']
191         dg = cache['dg']
192
193         # Initialize differentials along the network
194         delta = {}
195         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
196
197         for l in reversed(range(1, self.L + 1)):
198             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * drop
199
200     def update_parameters(self, params, learning_rate=0.1):
201         """
202         Parameters:
203         -----------
204         params : Dict[LinearParameters]
205             params[l].w = Weights
206             params[l].bias = Boolean
207             params[l].b = Bias
208         learning_rate : float
```

```
209             Default : 0.01
210
211         Returns:
212         --------
213         None
214         """
215         for param in params.values():
216             param.update(learning_rate)
217
218     def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000, print_cost
219         """
220         Parameters:
221         -----------
222         data : Dict[array_like]
223             data['x'] : array_like
224             data['y'] : array_like
225         learning_rate : float
226             Default : 0.1
227         lambda_ : float
228             Default : 0.0
229         num_iters : int
230             Default : 10000
231         print_cost_iter : int
232             Default: 1000   # 0 Doesn't print costs
233
234         Returns:
235         --------
236         costs : List[floats]
237         params : class[LinearParameters]
238         """
239         # Initialize parameters per layer
240         params = {}
241         for l in range(1, self.L + 1):
242             params[l] = LinearParameters(
243                 (self.nodes[l], self.nodes[l - 1]), self.bias[l])
244
245         # Initialize momentum
246         mom = Momentum(params, self.bias)
247
248         # Initialize batching
249         batching = ShuffleBatchData(data, self.batch_size)
250
251         costs = []
252         for i in range(num_iters):
253             batches = batching.get_batches()
254             for batch in batches:
255                 x = batch['x']
```

```
256                 y = batch['y']
257                 dropout = self.init_dropout(x.shape[1])
258                 cache = self.forward_propagation(params, x, dropout)
259                 cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
260                 costs.append(cost)
261                 self.backward_propagation(params, cache, y, dropout)
262                 mom.update(params, learning_rate)
263 
264             if (print_cost_iter != 0) and (i % print_cost_iter == 0):
265                 print(f'Cost after iteration {i}: {cost}')
266 
267         return params
268 
269     def evaluate(self, params, x):
270         """
271         Parameters:
272         -----------
273         params : Dict[LinearParameters]
274         x : array_like
275 
276         Returns:
277         --------
278         y_hat : array_like
279         """
280         cache = self.forward_propagation(params, x)
281         a = cache['a'][self.L]
282         y_hat = (~(a < 0.5)).astype(int)
283         return y_hat
284 
285     def accuracy(self, params, data):
286         """
287         Parameters:
288         -----------
289         params : Dict[LinearParameters]
290         data : Dict[array_like]
291             data['x'] : array_like
292             data['y'] : array_like
293 
294         Returns:
295         --------
296         accuracy : float
297         """
298         x = data['x']
299         y = data['y']
300 
301         y_hat = self.evaluate(params, x)
302         acc = np.sum(y_hat == y) / y.shape[1]
```

```
303
304         return acc
```

7.3 Root Mean Squared Propagation (RMSProp)

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

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

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

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

$$w = w - \alpha \frac{1}{\sqrt{s}} \odot \frac{\partial \mathbb{J}}{\partial w},$$

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

1. Initialize our parameters $w^{\{0\}}$ and $b^{\{0\}}$. Initialize ${s_w}^{\{0\}} = {s_b}^{\{0\}} = 0$. Fix a momentum $\beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small ($\epsilon = 10^{-8}$ is a good starting point).

2. For $0 \leq i <$ `num_iter`:

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

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

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

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

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

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

iv. Define

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

v. Update parameters via

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

7.3.1 Python Implementation via `numpy`

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation

class RMSProp():
    def __init__(self, params, bias, beta2=0.9, eps=1e-8):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w : array_like
            params[l].b : array_like
        bias : List[Boolean]
```

```
        beta2 : float
            Default: 0.9
        eps : float
            Default: 10^{-8}

        Returns:
        None
        """
        self.beta2 = beta2
        self.eps = eps
        self.bias = bias
        self.w = {}
        self.b = {}
        for l, param in params.items():
            self.w[l] = np.zeros(param.w.shape)
            if self.bias[l]:
                self.b[l] = np.zeros(param.b.shape)

    def update(self, params, learning_rate=0.01):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].dw : array_like
            params[l].db : array_like
        learning_rate : float
            Default: 0.01

        Returns:
        None
        """
        for l, param in params.items():
            sw = self.beta2 * self.w[l] + (1 - self.beta2) * (param.dw ** 2)
            self.w[l] = sw
            w = param.w - learning_rate * (param.dw / (np.sqrt(self.w[l]) + self.eps
            param.w = w
            if self.bias[l]:
                sb = self.beta2 * self.b[l] + (1 - self.beta2) * (param.db ** 2)
                self.b[l] = sb
                b = param.b - learning_rate * (param.db / (np.sqrt(self.b[l]) + self
                param.b = b

class NeuralNetwork():
    def __init__(self, config):
        """
        Parameters:
        -----------
```

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

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

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

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

        return D

    def forward_propagation(self, params, x, dropout=None):
        """
        Parameters:
        -----------
```

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

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

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

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

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

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

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

        Returns:
        --------
        cost: float
        """
        n = y.shape[1]
```

```
157         if self.lp_reg == 0:
158             lambda_ = 0.0
159 
160         # Compute regularization term
161         R = 0
162         for param in params.values():
163             R += np.sum(np.abs(param.w) ** self.lp_reg)
164         R *= (lambda_ / (2 * n))
165 
166         # Compute unregularized cost
167         a = np.clip(a, eps, 1 - eps)    # Bound a for stability
168         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
169 
170         cost = float(np.squeeze(J + R))
171 
172         return cost
173 
174     def backward_propagation(self, params, cache, y, dropout):
175         """
176         Parameters:
177         -----------
178         params : Dict[LinearParameters]
179             params[l].w = Weights
180             params[l].bias = Boolean
181             params[l].b = Bias
182         cache : Dict[array_like]
183             cache['a'] : array_like
184             cache['dg'] : array_like
185         y : array_like
186 
187         Returns:
188         --------
189         None
190         """
191 
192         # Retrieve cache
193         a = cache['a']
194         dg = cache['dg']
195 
196         # Initialize differentials along the network
197         delta = {}
198         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
199 
200         for l in reversed(range(1, self.L + 1)):
201             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * drop
202 
203     def update_parameters(self, params, learning_rate=0.1):
```

```
204         """
205         Parameters:
206         -----------
207         params : Dict[LinearParameters]
208             params[l].w = Weights
209             params[l].bias = Boolean
210             params[l].b = Bias
211         learning_rate : float
212             Default : 0.01
213 
214         Returns:
215         --------
216         None
217         """
218         for param in params.values():
219             param.update(learning_rate)
220 
221     def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000, print_cost
222         """
223         Parameters:
224         -----------
225         data : Dict[array_like]
226             data['x'] : array_like
227             data['y'] : array_like
228         learning_rate : float
229             Default : 0.1
230         lambda_ : float
231             Default : 0.0
232         num_iters : int
233             Default : 10000
234         print_cost_iter : int
235             Default: 1000   # 0 Doesn't print costs
236 
237         Returns:
238         --------
239         costs : List[floats]
240         params : class[LinearParameters]
241         """
242         # Initialize parameters per layer
243         params = {}
244         for l in range(1, self.L + 1):
245             params[l] = LinearParameters(
246                 (self.nodes[l], self.nodes[l - 1]), self.bias[l])
247 
248         # Initialize RMSProp
249         rms_prop = RMSProp(params, self.bias)
250 
```

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

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

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

        return params

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

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

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

        Returns:
```

```
        --------
        accuracy : float
        """
        x = data['x']
        y = data['y']

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

        return acc
```

7.4 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

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

7.4.1 Python Implementation via `numpy`

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation


class RMSProp():
    def __init__(self, params, bias, beta2=0.9, eps=1e-8):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w : array_like
            params[l].b : array_like
        bias : List[Boolean]
        beta2 : float
            Default: 0.9
        eps : float
            Default: 10^{-8}

        Returns:
        None
        """
        self.beta2 = beta2
        self.eps = eps
        self.bias = bias
        self.w = {}
        self.b = {}
        for l, param in params.items():
            self.w[l] = np.zeros(param.w.shape)
            if self.bias[l]:
                self.b[l] = np.zeros(param.b.shape)

    def update(self, params, learning_rate=0.01):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].dw : array_like
            params[l].db : array_like
        learning_rate : float
            Default: 0.01

        Returns:
```

```
        None
        """
        for l, param in params.items():
            sw = self.beta2 * self.w[l] + (1 - self.beta2) * (param.dw ** 2)
            self.w[l] = sw
            w = param.w - learning_rate * \
                (param.dw / (np.sqrt(self.w[l]) + self.eps))
            param.w = w
            if self.bias[l]:
                sb = self.beta2 * self.b[l] + \
                    (1 - self.beta2) * (param.db ** 2)
                self.b[l] = sb
                b = param.b - learning_rate * \
                    (param.db / (np.sqrt(self.b[l]) + self.eps))
                param.b = b


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

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

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

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

```
            if dropout != None:
                a[l] = dropout[l] * a[l]

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

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

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

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

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

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

        return cost

    def backward_propagation(self, params, cache, y, dropout):
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
            params[l].w = Weights
            params[l].bias = Boolean
```

```
            params[l].b = Bias
        cache : Dict[array_like]
            cache['a'] : array_like
            cache['dg'] : array_like
        y : array_like

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

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

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

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

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

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

    def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000, print_cost
        """
        Parameters:
        -----------
        data : Dict[array_like]
            data['x'] : array_like
```

```
            data['y'] : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000
        print_cost_iter : int
            Default: 1000   # 0 Doesn't print costs

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

        # Initialize RMSProp
        rms_prop = RMSProp(params, self.bias)

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

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

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

        return params

    def evaluate(self, params, x):
```

```
        """
        Parameters:
        -----------
        params : Dict[LinearParameters]
        x : array_like

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

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

        Returns:
        --------
        accuracy : float
```

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

```
import copy

import numpy as np
from sklearn.utils import shuffle

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
    """
    ## Retrieve accelerations and gradients
    sw = s['w']
    sb = s['b']
    dw = grads['w']
    db = grads['b']
    L = len(dw)

    for l in range(1, L + 1):
        sw[l] = beta2 * sw[l] + (1 - beta2) * (dw[l] * dw[l])
        sw[l] /= (1 - beta2 ** update_iter)
        assert(sw[l].shape == dw[l].shape)
        sb[l] = beta2 * sb[l] + (1 - beta2) * (db[l] * db[l])
        sb[l] /= (1 - beta2 ** update_iter)
        assert(sb[l].shape == db[l].shape)

    s = {'w' : sw, 'b' : sb}
    return s


def update_parameters_adam(params, grads, epoch, batch_iter, v, s, momenta=[1e-8, 0.
    """
    Parameters
    ----------
    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)
    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
    epoch : int
    batch_iter : int
    learning_rate : float
        Default: 0.01
    momenta : List[float]
        momenta[0] = epsilon
```

```
            Default: 10^{-8}
        momenta[1] = beta_1
            Default: 0.9
        momenta[2] = beta_2
            Default: 0.999

    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]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    batch_size : int
    lambda_ : float
        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
    """
    n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
    params = utils.initialize_parameters_random(layers)
    v, s = initialize_momenta(layers)


    ## main descent loop
    for i in range(num_iters):
        batches = get_batches(x, y, batch_size)
        ## batch loop
        batch_iter = 1
        cost = 0
        for batch in batches:
            x = batch['x']
            y = batch['y']
            cache = utils.forward_propagation(x, params, activators)
            cost += utils.compute_cost(y, params, cache)
```

```
281             grads = utils.backward_propagation(x, y, params, cache, activators)
282             params =  update_parameters_adam(params,
283                                              grads,
284                                              i,
285                                              batch_iter,
286                                              v,
287                                              s,
288                                              momenta=[1e-8, 0.9, 0.999],
289                                              learning_rate=0.01,
290                                              decay_rate = 0.0)
291             batch_iter += 1
292 
293         if print_cost and i % 1000 == 0:
294             print(f'Cost after iteration {i}: {cost}')
295 
296     return params, cost
```

8 Tuning Hyper-Parameters

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

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

Furthermore, suppose we impose a neural network architecture which has a collection of hyper-parameters (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 [7].

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 **??**

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{m_0} \end{bmatrix}}_{\text{Layer } 0} \xrightarrow{\Phi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]m_1} \end{bmatrix} \xrightarrow{BN_{\gamma_1,\beta}} \begin{bmatrix} \tilde{z}^{[\ell]1} \\ \vdots \\ \tilde{z}^{[\ell]}{}_{m_\ell} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]m_1} \end{bmatrix}}_{\text{Layer } 1} \xrightarrow{\Phi^{[2]}} \cdots$$

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

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

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

Proof: We compute directly after noting that

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

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

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

as desired. □

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.,*

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

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

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

denote

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

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)

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

  and hence

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

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

  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

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

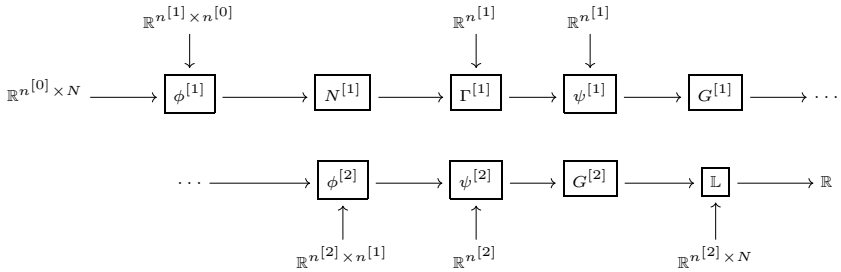
iv. Update parameters.

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

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

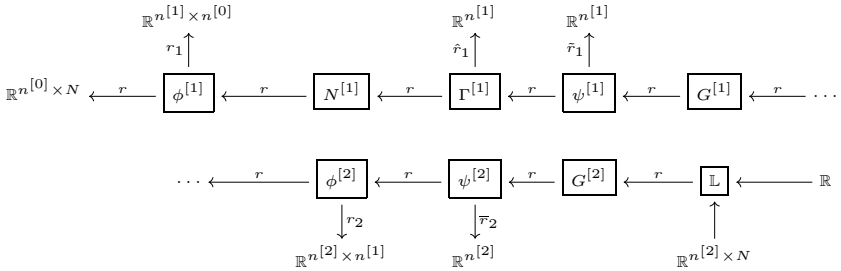
   where

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

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

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

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

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

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

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

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

and so

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

Finally, we conclude that

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

and

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

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

Part III
Convolutional Neural Networks

11 An Introduction to Convolutions

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

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

and 10 represents "white" and 0 represents "black".

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

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

11.1 Cross-Correlation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 11.1. *Suppose*

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

and

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

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

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

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

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

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

and hence

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

Example 11.2. *Suppose*

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

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

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

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

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

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

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

11.2 Convolution with Padding

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

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

Example 11.3. *Suppose*

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

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

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

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

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

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

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

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

and we write

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

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

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

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

11.3 Strided Convolution

We note that in our definition of a convolution

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

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

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

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

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

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

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

Example 11.4. *Suppose*

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

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

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

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

and hence that

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

or rather

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

11.4 Strided Convolutions with Padding

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

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

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

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

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

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

and now we immediately see by chasing the definitions that

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

Example 11.5. *Suppose*

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

and we have a filter

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

We first compute $F *_2^1 x$: Since we we're using a padding of $p = 1$, we have that

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

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

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

that is, $F *_2^1 x \in \mathbb{R}^{2\times 2}$. We now compute

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

or rather

$$F *_2^1 x = \begin{bmatrix} 1 & 2 \\ 4 & 0 \end{bmatrix}.$$

11.5 Convolutions Over Volumes

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

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

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

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

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

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

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

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

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

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

11.6 Multiple Filters

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

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

Then we have that

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

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

$$ (F *^p_s x)_\eta{}^k{}_l = (F_\eta *^p_s x)^k{}_l. $$

12 Convolutional Networks

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

12.1 Convolutional Layers (conv)

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

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

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

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

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

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

Letting

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

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

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

where

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

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

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

plus the bias terms

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

that is,

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

12.2 Pooling Layers (`pool`)

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

12.2.1 Max Pooling

Suppose

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

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

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

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

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

12.2.2 Average Pooling

Suppose

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

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

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

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

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

12.3 A Convolutional Network

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

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

We similarly use $\mathsf{pool}(1)$ to encompass the following information:

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

This yields a network architecture of the following form:

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

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

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

12.4 Backpropagation

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

$$x = (x^\rho{}_{ij})$$

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

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

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

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

is given by

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

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

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

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

and hence

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

Next we consider

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

and hence

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

Finally, we consider

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

and hence

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

Appendices

A utils.py

```
#! python3
import copy

import numpy as np
from sklearn.utils import shuffle

import mlLib.npActivators as npActivators
from mlLib.npActivators import ACTIVATORS

## Usefule printing function
def print_array_dict(D):
    """
    Parameters
    ----------
    D : Dict[array_like]

    Returns
    -------
    None
    """
    txt = "Array {0} has shape {1}\n{2}"
    for k, v in D.items():
        print(txt.format(str(k), v.shape, v))


## Partition data into training, development, and test sets
def partition_data(x, y, train_ratio):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, N)
    y : array_like
        y.shape = (k, N)
    train_ratio : float
        0<=train_ratio<=1

    Returns
    -------
    train : Tuple[array_like]
    dev : Tuple[array_like]
```

```
    test : Tuple[array_like]
    """
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
    x = x.T
    y = y.T

    ## Get the size of partitions
    N = x.shape[1]
    N_train = int(train_ratio * N)
    N_mid = (N - N_train) // 2

    ## Create partitions
    train = (x[:,:N_train], y[:,:N_train])
    dev = (x[:,N_train:N_train + N_mid], y[:,N_train:N_train + N_mid])
    test = (x[:,N_train + N_mid:], y[:,N_train + N_mid:])

    assert(x.all() == np.concatenate([train[0], dev[0], test[0]], axis=1).all())
    assert(y.all() == np.concatenate([train[1], dev[1], test[1]], axis=1).all())

    return train, dev, test

## Partition training data into batches
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
    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))]
```

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

```
136         b[l] : array_like
137             dbl.shape = (layers[l], 1)
138     """
139     w = {}
140     b = {}
141     for l in range(1, len(layers)):
142         w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
143         b[l] = np.zeros((layers[l], 1))
144     params = {'w' : w, 'b' : b}
145     return params
146 
147 ## Forward and Backward Linear Activations
148 def linear_activation_forward(a_prev, w, b, activator):
149     """
150     Parameters
151     ----------
152     a_prev : array_like
153         a_prev.shape = (layers[l], n)
154     w : array_like
155         w.shape = (layers[l+1], layers[l])
156     b : array_like
157         b.shape = (layers[l+1], 1)
158     activator : str
159         activator in ACTIVATORS
160 
161     Returns
162     -------
163     z : array_like
164         z.shape = (layer_dims[l+1], n)
165     a : array_like
166         a.shape = (layer_dims[l+1], n)
167     """
168     assert activator in ACTIVATORS, f'{activator} is not a valid activator.'
169 
170     z = w @ a_prev + b
171     if activator == 'relu':
172         a, _ = npActivators.relu(z)
173     elif activator == 'sigmoid':
174         a, _ = npActivators.sigmoid(z)
175     elif activator == 'tanh':
176         a, _ = npActivators.tanh(z)
177     return z, a
178 
179 def linear_activation_backward(delta_next, z, w, activator):
180     """
181     Parameters
182     ----------
```

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

```
    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_dropout(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)
        D[l] = a Boolean array astype(int)
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    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 = {}
```

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

```
        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 = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activators
        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 = {'w' : dw, 'b' : db}
    return grads


## Forward and Backward Propagation with L2-Regularization
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]
```

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

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

```
        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


## Update parameters via gradient descent
def update_parameters(params, grads, learning_rate=0.01):
    """
    Parameters
    ----------
    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)
    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
    learning_rate : float
        Default: 0.01
        The learning rate for gradient descent

    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)
    """
    ## Retrieve parameters
    w = copy.deepcopy(params['w'])
    b = copy.deepcopy(params['b'])
    L = len(w)

    ## Retrieve gradients
    dw = grads['w']
    db = grads['b']

    ## Perform update
    for l in range(1, L + 1):
        w[l] = w[l] - learning_rate * dw[l]
        b[l] = b[l] - learning_rate * db[l]

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









#### Dropout NN Model ####
def model_nn(x, y, hidden_layer_sizes, activators, keep_prob=1.0, num_iters=10000, p
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
```

```
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        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
    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
    """
    ## Set dimensions and Initialize parameters
    n, layers = dim_retrieval(x, y, hidden_layer_sizes)
    params = 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 = compute_cost(cache, y)
        grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
        params = update_parameters(params, grads)

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

    return params, cost
```

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

```

#######
if __name__ == '__main__':
    test_dropout_nn()
```

B activators.py

```
import numpy as np

ACTIVATORS = ['relu', 'sigmoid', 'tanh', 'linear', 'softmax']

## Activator functions
# The (leaky-)ReLU function
def relu(z, beta=0.0):
    """
    Parameters
    ----------
    z : array_like
    beta : float

    Returns
    -------
    r : array_like
        The (broadcasted) ReLU function when beta=0, the leaky-ReLU otherwise.
    dr : array_like
        The (broadcasted) derivative of the (leaky-)ReLU function
    """
    # Change scalar to array if needed
    z = np.array(z)
    # Compute value of ReLU(z)
    r = np.maximum(z, beta * z)
    # Compute differential ReLU'(z)
    dr = ((~(z < 0)) * 1) + ((z < 0) * beta)
    return r, dr

# The sigmoid function
def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
        The (broadcasted) value of the sigmoid function evaluated at z
```

```
40      dsigma : array_like
41          The (broadcasted) derivative of the sigmoid function evaluate at z
42      """
43      # Compute value of sigmoid
44      sigma = (1 / (1 + np.exp(-z)))
45      # Compute differential of sigmoid
46      dsigma = sigma * (1 - sigma)
47      return sigma, dsigma
48
49  # The hyperbolic tangent function
50  def tanh(z):
51      """
52      Parameters
53      ----------
54      z : array_like
55
56      Returns
57      phi : array_like
58          The (broadcasted) value of the hyperbolic tangent function evaluated at z
59      dphi : array_like
60          The (broadcasted) derivative of hyperbolic tangent function evaluated at z
61      """
62      # Compute value of tanh
63      phi = np.tanh(z)
64      # Compute differential of tanh
65      dphi = 1 - (phi * phi)
66      return phi, dphi
67
68  # The linear activator function
69  def linear(z):
70      """
71      Parameters
72      ----------
73      z : array_like
74
75      Returns
76      -------
77      id : array_like
78      d_id
79      """
80      id = z
81      d_id = np.ones(z.shape)
82      return id, d_id
```

C The Reverse Differential

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

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

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

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

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

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

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

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

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

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

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

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

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

It then follows that

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

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

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

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

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

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

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

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

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

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

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

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

and hence that

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

Proposition C.5. *Suppose we have the compositional diagram*

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

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

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

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

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

as desired. □

The following needs to be refined further still.

Example C.6. *Suppose* $f : (\mathbb{R}^{m\times n}, (X^i_j), F) \to (\mathbb{R}, (t), \delta)$ *is smooth, where* F *is the Frobenius inner product. Suppose* $v \in T_P\mathbb{R}^{m\times n} \leftrightsquigarrow V \in \mathbb{R}^{m\times n}$ *are represented via*

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

and in coordinates, we have that

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

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

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

It then follows that

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

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

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

evaluates to

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

and hence that

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

Lemma C.7. *Suppose $f : \mathbb{R}^{n\times m} \to \mathbb{R}^k$, and for $P \in \mathbb{R}^{n\times m}$, let $R = rf_P$. Then $R \in \mathbb{R}^k{}_n{}^m$ is rank $(1,2)$-tensor written in coordinates as*

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

and the components is given by

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

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

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

as desired. □

References

[1] Henk P Barendregt and Erik Barendsen. Introduction to lambda calculus. In *Aspens Workshop on Implementation of Functional Languages, Göteborg. Programming Methodology Group, University of Göteborg and Chalmers University of Technology*, volume 85, 1988.

[2] Richard F Blute, J Robin B Cockett, and Robert AG Seely. Cartesian differential categories. *Theory and Applications of Categories*, 22(23):622–672, 2009.

[3] Robin Cockett, Geoffrey Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon Plotkin, and Dorette Pronk. Reverse derivative categories. *arXiv preprint arXiv:1910.07065*, 2019.

[4] Geoffrey SH Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In *European Symposium on Programming*, pages 1–28. Springer, Cham, 2022.

[5] Brendan Fong, David Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

[6] Bruno Gavranović. Compositional deep learning. *arXiv preprint arXiv:1907.08292*, 2019.

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

[8] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[9] Carol Mak and Luke Ong. A differential-form pullback programming language for higher-order reverse-mode automatic differentiation. *arXiv preprint arXiv:2002.08241*, 2020.

[10] Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010.

[11] Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category theory in machine learning. *arXiv preprint arXiv:2106.07032*, 2021.

[12] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.