

Neural Networks

Matt R

July 8, 2022

Contents

I	Neural Networks and Deep Learning	4
1	Logistic Regression	5
1.1	The Gradient	6
1.2	Implementation in Python via <code>numpy</code>	9
1.3	Implementation in Python via <code>sklearn</code>	13
2	Neural Networks: A Single Hidden Layer	15
2.1	Activation Functions	17
2.1.1	The Sigmoid Function	17
2.1.2	The Hyperbolic Tangent Function	18
2.1.3	The Rectified Linear Unit Function	18
2.1.4	The Softmax Function	19
2.2	Backward Propagation	20
3	Deep Neural Networks	26
3.1	Implementation in Python via <code>numpy</code>	28
3.2	Implementation in Python via <code>tensorflow</code>	32
II	Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization	35
4	Training, Development and Test Sets	36
4.1	Python Implementation	38

5	Regularization	41
5.1	(Inverted) Dropout Regularization	42
5.1.1	Python Implementation	43
5.2	Data Augmentation	48
5.3	Early Stopping	49
6	Gradients and Numerical Remarks	50
6.1	Numerical Gradient Checking	50
6.2	Python Implementation	51
7	Gradient Descent	53
7.0.1	Python Implementation via <code>numpy</code>	55
7.1	Weighted Averages	61
7.2	Gradient Descent with Momentum	64
7.2.1	Python Implementation via <code>numpy</code>	65
7.3	Root Mean Squared Propagation (RMSProp)	72
7.3.1	Python Implementation via <code>numpy</code>	73
7.4	Adaptive Moment Estimation: The Adam Algorithm	80
7.4.1	Python Implementation via <code>numpy</code>	81
7.5	Learning Rate Decay	88
7.6	Python Implementation	88
8	Tuning Hyper-Parameters	96
8.1	Python Implementation	97
9	Batch Normalization	98
9.1	Backward Propagation	100
9.2	Inferencing	106
9.3	Algorithm Outline	107
9.4	Better Backpropagation	109
9.5	Python Implementation	115
10	Multi-Class Softmax Regression	116
III	Convolutional Neural Networks	120
11	An Introduction to Convolutions	121
11.1	Cross-Correlation	121
11.2	Convolution with Padding	123

11.3 Strided Convolution	125
11.4 Strided Convolutions with Padding	126
11.5 Convolutions Over Volumes	127
11.6 Multiple Filters	128
12 Convolutional Networks	129
12.1 Convolutional Layers (<code>conv</code>)	129
12.2 Pooling Layers (<code>pool</code>)	130
12.2.1 Max Pooling	130
12.2.2 Average Pooling	130
12.3 A Convolutional Network	131
12.4 Backpropagation	132
Appendix A <code>utils.py</code>	134
Appendix B <code>activators.py</code>	148
Appendix C The Reverse Differential	150
References	155

Part I

Neural Networks and Deep Learning

1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have training examples $x \in \mathbb{R}^{n \times N}$ with binary labels $y \in \{0, 1\}^{1 \times N}$. We desire to train a model which yields an output a which represents

$$a = \mathbb{P}(y = 1|x).$$

To this end, let $\sigma : \mathbb{R} \rightarrow (0, 1)$ denote the sigmoid function, i.e.,

$$\sigma(z) = \frac{1}{1 + e^{-z}},$$

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

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

To analyze the accuracy of model, we need a way to compare y and a , and ideally this functional comparison can be optimized with respect to (w, b) in such a way to minimize an error. To this end, we note that

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

or rather

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

so $\mathbb{P}(y|x)$ represents the *corrected probability*. Now since we want

$$a \approx 1 \quad \text{when } y = 1,$$

and

$$a \approx 0 \quad \text{when } y = 0,$$

and $0 \leq a \leq 1$, any error using differences won't be refined enough to analyze when tuning the model. Moreover, since introducing the sigmoid function, our usual mean-squared-error function won't be convex. This leads us to apply the log function, which when restricted to $(0, 1)$ is a bijective mapping of $(0, 1) \rightarrow (-\infty, 0)$. This leads us to define our log-loss function

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

and finally, since we wish to analyze how our model performs on the entire training set, we need to average our log-loss functions to obtain our cost function \mathbb{J} defined by

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

1.1 The Gradient

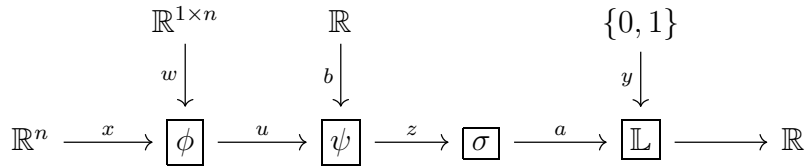
We wish to compute the gradient of our cost function \mathbb{J} with respect to our trainable parameters, $w \in \mathbb{R}^{1 \times n}$ and $b \in \mathbb{R}$. To this end, we define the functions

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

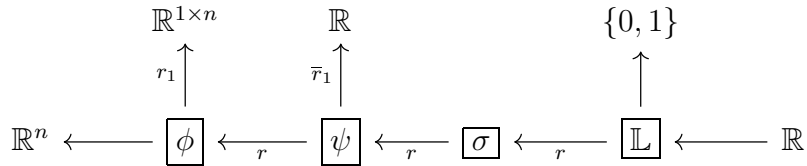
and

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

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



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



1.

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

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

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

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

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

and hence that

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

2.

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

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

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

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

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

We then immediately have that

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

and

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

3.

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

Then

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

4.

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

Then

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

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

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

and

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

1.2 Implementation in Python via numpy

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

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

```

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

```

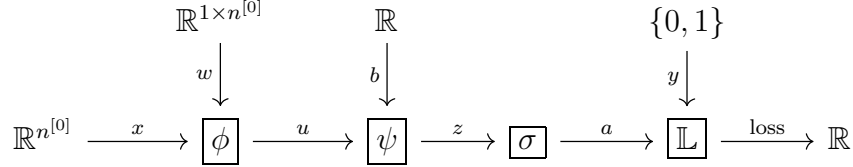
```

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

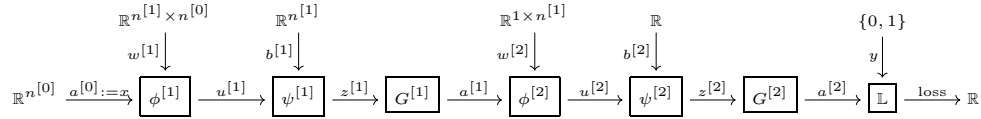
```

2 Neural Networks: A Single Hidden Layer

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



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



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

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

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

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

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

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

castingDifferential

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

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

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

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

We use the notation

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

and thus may write

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

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

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

Proof: We calculate

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

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

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

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

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

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

and the result follows. \square

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

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

2.1 Activation Functions

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

2.1.1 The Sigmoid Function

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

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

We note that since

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

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

2.1.2 The Hyperbolic Tangent Function

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

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

We then calculate

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

Furthermore, we note that

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

Indeed,

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

as desired.

2.1.3 The Rectified Linear Unit Function

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

$$\text{ReLU} : \mathbb{R} \rightarrow \mathbb{R}, \quad \text{ReLU}(z; \beta) = \max\{\beta z, z\},$$

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

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

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

We then calculate

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

where

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

is the indicator function.

2.1.4 The Softmax Function

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

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

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

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

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

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

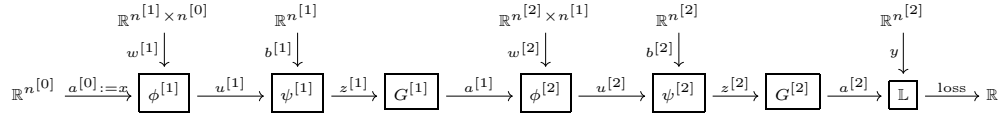
That is, as a map $dS_z : T_z \mathbb{R}^m \rightarrow T_{S(z)} \mathbb{R}^m$, we have that

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

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

2.2 Backward Propagation

We consider a neural network of the form



where we have the functions:

1.

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

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

2.

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

is given by

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

3.

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

is given by

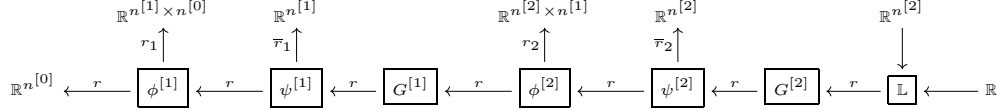
$$\psi^{[\ell]}(b, x) = x + b.$$

4.

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

is the given loss-function.

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



First, we need to consider our individual derivatives:

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

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

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

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

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

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

Then we have two differentials to consider:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and we have our cost function

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

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

•

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

•

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

•

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

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

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

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

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

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

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

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

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

and hence

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

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

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

and hence

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

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

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

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

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

and hence

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

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

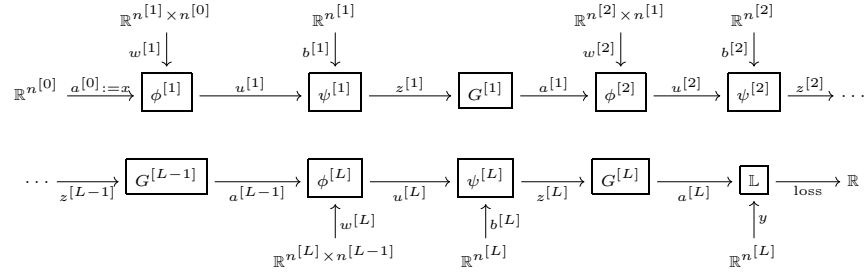
$$\begin{aligned} r_1 F_p &= r_1 (\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\ &= r_1 \phi_p^{[1]} \circ r\psi_{u^{[1]}}^{[1]} \circ rG_{z^{[1]}}^{[1]} \circ r\phi_{a^{[1]}}^{[2]} \circ r\psi_{u^{[2]}}^{[2]} \circ rG_{z^{[2]}}^{[2]}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

and hence

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

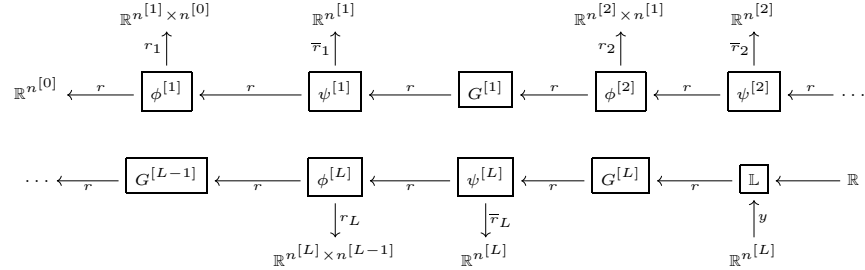
3 Deep Neural Networks

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



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

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



We compute differentials recursively as follows:

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

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

2. Compute

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

and

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

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

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

4. Compute

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

and

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

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

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

6. Compute

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

and

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

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

3.1 Implementation in Python via numpy

We implement a neural network with an arbitrary number of layers and nodes, with the ReLU function as the activator on all hidden nodes and the sigmoid function on the output layer for binary classification with the log-loss function.

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

```

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

```

```

135         None
136         """
137         for param in params.values():
138             param.update(learning_rate)
139
140     def fit(self, x, y, learning_rate=0.1, lambda_=0.01, num_iters=10000):
141         """
142         Parameters:
143         -----
144         x : array_like
145         y : array_like
146         learning_rate : float
147             Default : 0.1
148         lambda_ : float
149             Default : 0.0
150         num_iters : int
151             Default : 10000
152
153         Returns:
154         -----
155         costs : List[floats]
156         params : class[Parameters]
157         """
158         # Initialize parameters per layer
159         params = {}
160         for l in range(1, self.L + 1):
161             params[l] = LinearParameters((self.nodes[l], self.nodes[l - 1]), self.b)
162
163         costs = []
164         for i in range(num_iters):
165             cache = self.forward_propagation(params, x)
166             cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
167             costs.append(cost)
168             self.backward_propagation(params, cache, y)
169             self.update_parameters(params, learning_rate)
170
171             if i % 1000 == 0:
172                 print(f'Cost_after_iteration_{i}:_{cost}')
173
174         return params
175
176     def evaluate(self, params, x):
177         """
178         Parameters:
179         -----
180         params : class[Parameters]
181         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}

```



```

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

```

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

Part II

Improving Deep Neural Networks: Hyperparameter Tuning, Regularization, and Optimization

4 Training, Development and Test Sets

Let $\mathbb{D} = \{(x_j, y_j) \in \mathbb{R}^m \times \mathbb{R}^K : 1 \leq j \leq N\}$ denote a dataset. Then we partition \mathbb{D} into three distinct sets

$$\mathbb{D} = \mathbb{X} + \mathcal{D} + \mathcal{T},$$

where \mathbb{X} is called our *training set*, \mathcal{D} is called our *development, or cross-validation set*, and \mathcal{T} is called our *test set*. We make this partition randomly, however, if $N = |\mathbb{D}| \leq 10^4$, we see a partition being divided accordingly to the following ratios:

$$n_X := |\mathbb{X}| \approx \frac{3}{5}N,$$

$$n_D := |\mathcal{D}| \approx \frac{1}{5}N,$$

and

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

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

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

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

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

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

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

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

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

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

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

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

A methodology for using errors could be as follows

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

4.1 Python Implementation

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

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

```

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

```

```

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

```


5 Regularization

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

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

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

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

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

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

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

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

and

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

The idea behind regularization is that we're now minimizing

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

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

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

- i. Partition our dataset $\mathbb{D} = \mathbb{X} \cup \mathcal{D} \cup \mathcal{T}$.
- ii. Give a set Λ of potential regularization parameters.
- iii. For each $\lambda \in \Lambda$, we first train on \mathbb{X} , that is, we obtain

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

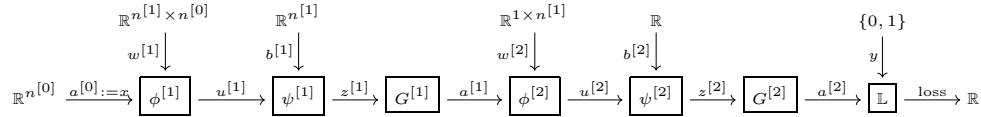
$$= \arg \min_{w, b} \left\{ \frac{1}{n_X} \sum_{(x, y) \in \mathbb{X}} \mathbb{L}(a^{[L]}, y) + \frac{\lambda}{2n_X} \sum_{\ell=1}^L \|w^{[\ell]}\|_F^2 \right\}$$

which is dependent on λ .

- iv. Then using the aforementioned $(w, b) = (w, b)(\lambda)$, we evaluate $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$.
- v. After finding $\mathcal{E}_\lambda(\mathbb{X})$ and $\mathcal{E}_\lambda(\mathcal{D})$ for each $\lambda \in \Lambda$, we choose our desired λ and hence our desired parameters w and b .
- vi. We evaluate our model on \mathcal{T} to determine the overall accuracy.

5.1 (Inverted) Dropout Regularization

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



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

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

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

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

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

where

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

- ii. During forward propagation, we redefine

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

- iii. During backward propagation, we define

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

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

5.1.1 Python Implementation

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

5.2 Data Augmentation

This section requires work.

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

5.3 Early Stopping

This section requires work.

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

6 Gradients and Numerical Remarks

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

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

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

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

6.1 Numerical Gradient Checking

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

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

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

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

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

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

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

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

6.2 Python Implementation

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

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

```
1 ## f(x) = x_1*x_2*...*x_n
2 def fctn(x):
3     n = x.shape[0]
4     y = np.prod(x)
5     grad = np.zeros((n, 1))
6     for i in range(n):
7         omit = 1 - np.eye(1, n, i).T
8         omit = np.array(omit, dtype=bool)
9         grad[i, 0] = np.prod(x, where=omit)
10    return y, grad
11
12 def gradient_check(grad, f, x, epsilon=1e-3):
13     """
14     Parameters
15     -----
16     grad : array_like
17         grad.shape= (n, 1)
18     f : function
19         The function to check.
20     x : array_like
21         x.shape = (n, 1)
22     epsilon : float
23         Default 0.001
24     Returns
25     error : float
26     -----
27     """
```

```

28     n = x.shape[0]
29     y_diffs = []
30     for i in range(n):
31         e = np.eye(1, n, i).T
32         x_plus = x + epsilon * e
33         x_minus = x - epsilon * e
34         y_plus, _ = f(x_plus)
35         y_minus, _ = f(x_minus)
36         y_diffs.append(y_plus - y_minus)
37     y_diffs = np.array(y_diffs).reshape(n, 1)
38     y_diffs = y_diffs / (2 * epsilon)
39
40     error = (np.linalg.norm(y_diffs - grad)
41             / (np.linalg.norm(y_diffs) + np.linalg.norm(grad)))
42     return error

```

7 Gradient Descent

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

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

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

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

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

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

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

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

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

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

iv. Perform gradient descent:

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

We make several remarks about mini-batch gradient descent:

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

$$b = 2^p,$$

for some $p \in \{6, 7, 8, 9, 10\}$ (usually $p < 10$).

- Choose a batch size b that ensures your computer's CPU/GPU can hold a dataset of that size.

7.0.1 Python Implementation via numpy

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

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

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

7.1 Weighted Averages

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

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

has variable T .

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and so

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

We then see that

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

which is the explicit definition of a weighted-average.

7.2 Gradient Descent with Momentum

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

We first recall our gradient descent algorithm:

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

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

- iv. We update parameters

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

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

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

- i. Apply forward propagation on \mathbb{X}^k .
- ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
- iii. Apply backward propagation on \mathbb{X}^k to obtain

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

- iv. Define

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

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

- v. We update parameters

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

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

7.2.1 Python Implementation via `numpy`

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

```

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

```

```

21         """
22         self.beta1 = beta1
23         self.bias = bias
24         self.w = {}
25         self.b = {}
26         for l, param in params.items():
27             self.w[l] = np.zeros(param.w.shape)
28             if self.bias[l]:
29                 self.b[l] = np.zeros(param.b.shape)
30
31     def update(self, params, learning_rate=0.01, update_params=True):
32         """
33         Parameters:
34         -----
35         params : Dict[LinearParameters]
36             params[l].dw : array_like
37             params[l].db : array_like
38         learning_rate : float
39             Default: 0.01
40         update_params : Boolean
41             Default: True
42
43         Returns:
44         None
45         """
46         for l, param in params.items():
47             vw = self.beta1 * self.w[l] + (1 - self.beta1) * param.dw
48             self.w[l] = vw
49             if update_params:
50                 w = param.w - learning_rate * self.w[l]
51                 param.w = w
52             if self.bias[l]:
53                 vb = self.beta1 * self.b[l] + (1 - self.beta1) * param.db
54                 self.b[l] = vb
55                 if update_params:
56                     b = param.b - learning_rate * self.b[l]
57                     param.b = b
58
59     class NeuralNetwork():
60         def __init__(self, config):
61             """
62             Parameters:
63             -----
64             config : Dict
65                 config['lp_reg'] = 0,1,2
66                 config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
67                 config['nodes'] = List[int]

```

```

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

```

```

115         x : array_like
116
117     Returns:
118     -----
119     cache = Dict[array_like]
120         cache['a'] = a
121         cache['dg'] = dg
122
123     """
124     # Initialize dictionaries
125     a = {}
126     dg = {}
127
128     a[0], dg[0] = apply_activation(x, self.activators[0])
129     if dropout != None:
130         a[0] = dropout[0] * a[0]
131
132     for l in range(1, self.L + 1):
133         z = params[l].forward(a[l - 1])
134         a[l], dg[l] = apply_activation(z, self.activators[l])
135         if dropout != None:
136             a[l] = dropout[l] * a[l]
137
138     cache = {'a': a, 'dg': dg}
139     return cache
140
141 def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
142     """
143     Parameters:
144     -----
145     params: Dict[LinearParameters]
146     a: array_like
147     y: array_like
148     lambda_: float
149         Default: 0.01
150     eps: float
151         Default: 1e-8
152
153     Returns:
154     -----
155     cost: float
156     """
157     n = y.shape[1]
158     if self.lp_reg == 0:
159         lambda_ = 0.0
160
161     # Compute regularization term

```

```

162         R = 0
163         for param in params.values():
164             R += np.sum(np.abs(param.w) ** self.lp_reg)
165         R *= (lambda_ / (2 * n))
166
167         # Compute unregularized cost
168         a = np.clip(a, eps, 1 - eps) # Bound a for stability
169         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
170
171         cost = float(np.squeeze(J + R))
172
173         return cost
174
175     def backward_propagation(self, params, cache, y, dropout):
176         """
177         Parameters:
178         -----
179         params : Dict[LinearParameters]
180             params[l].w = Weights
181             params[l].bias = Boolean
182             params[l].b = Bias
183         cache : Dict[array_like]
184             cache['a'] : array_like
185             cache['dg'] : array_like
186         y : array_like
187
188         Returns:
189         -----
190         None
191         """
192
193         # Retrieve cache
194         a = cache['a']
195         dg = cache['dg']
196
197         # Initialize differentials along the network
198         delta = {}
199         delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
200
201         for l in reversed(range(1, self.L + 1)):
202             delta[l - 1] = dg[l - 1] * params[l].backward(delta[l], a[l - 1]) * dropout[l]
203
204     def update_parameters(self, params, learning_rate=0.1):
205         """
206         Parameters:
207         -----
208         params : Dict[LinearParameters]

```

```

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

```

```

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

```

```
y = data['y']
```

7.3 Root Mean Squared Propagation (RMSProp)

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

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

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

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

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

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

1. Initialize our parameters $w^{\{0\}}$ and $b^{\{0\}}$. Initialize $s_w^{\{0\}} = s_b^{\{0\}} = 0$. Fix a momentum $\beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small ($\epsilon = 10^{-8}$ is a good starting point).
2. For $0 \leq i < \text{num_iter}$:
 - a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$
 - b. For $1 \leq k \leq B$:

- i. Apply forward propagation on \mathbb{X}^k .
- ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
- iii. Apply backward propagation on \mathbb{X}^k to obtain

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

- iv. Define

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

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

- v. Update parameters via

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

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

7.3.1 Python Implementation via numpy

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

```

1 #! python3
2
3 import numpy as np
4
5 from mlLib.utils import LinearParameters, ShuffleBatchData, apply_activation
6
7 class RMSProp():
8     def __init__(self, params, bias, beta2=0.9, eps=1e-8):
9         """
10         Parameters:
11         -----
12         params : Dict[LinearParameters]
13             params[1].w : array_like
14             params[1].b : array_like
15         bias : List[Boolean]
16         beta2 : float
17             Default: 0.9

```

```

18         eps : float
19             Default: 10^{-8}
20
21     Returns:
22     None
23     """
24     self.beta2 = beta2
25     self.eps = eps
26     self.bias = bias
27     self.w = {}
28     self.b = {}
29     for l, param in params.items():
30         self.w[l] = np.zeros(param.w.shape)
31         if self.bias[l]:
32             self.b[l] = np.zeros(param.b.shape)
33
34     def update(self, params, learning_rate=0.01, update_params=True):
35         """
36         Parameters:
37         -----
38         params : Dict[LinearParameters]
39             params[l].dw : array_like
40             params[l].db : array_like
41         learning_rate : float
42             Default: 0.01
43         update_params : Boolean
44             Default: True
45
46         Returns:
47         None
48         """
49         for l, param in params.items():
50             sw = self.beta2 * self.w[l] + (1 - self.beta2) * (param.dw ** 2)
51             self.w[l] = sw
52             if update_params:
53                 w = param.w - learning_rate * (param.dw / (np.sqrt(self.w[l]) + self.eps))
54                 param.w = w
55             if self.bias[l]:
56                 sb = self.beta2 * self.b[l] + (1 - self.beta2) * (param.db ** 2)
57                 self.b[l] = sb
58                 if update_params:
59                     b = param.b - learning_rate * (param.db / (np.sqrt(self.b[l]) + self.eps))
60                     param.b = b
61
62     class NeuralNetwork():
63         def __init__(self, config):
64             """

```

```

65         Parameters:
66         -----
67         config : Dict
68             config['lp_reg'] = 0,1,2
69             config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
70             config['nodes'] = List[int]
71             config['bias'] = List[Boolean]
72             config['activators'] = List[str]
73             config['keep_probs'] = List[float]
74
75         Returns:
76         -----
77         None
78         """
79         self.config = config
80         self.lp_reg = config['lp_reg']
81         self.batch_size = config['batch_size']
82         self.nodes = config['nodes']
83         self.bias = config['bias']
84         self.activators = config['activators']
85         self.keep_probs = config['keep_probs']
86         self.L = len(config['nodes']) - 1
87
88     def init_dropout(self, num_examples, seed=101011):
89         """
90         Parameters:
91         -----
92         num_examples : int
93         seed : int
94             Default: 1 # For reproducibility
95
96         Returns:
97         -----
98         D : Dict[layer : array_like]
99         """
100         np.random.seed(seed)
101         D = {}
102         for l in range(self.L + 1):
103             D[l] = np.random.rand(self.nodes[l], num_examples)
104             D[l] = (D[l] < self.keep_probs[l]).astype(int)
105             D[l] = D[l] / self.keep_probs[l]
106             assert (D[l].shape == (self.nodes[l], num_examples)), "Dropout_matrices."
107
108         return D
109
110     def forward_propagation(self, params, x, dropout=None):
111         """

```

```

112     Parameters:
113     -----
114     params : Dict[class[Parameters]]
115             params[l].w = Weights
116             params[l].bias = Boolean
117             params[l].b = Bias
118     x : array_like
119
120     Returns:
121     -----
122     cache = Dict[array_like]
123             cache['a'] = a
124             cache['dg'] = dg
125
126     """
127     # Initialize dictionaries
128     a = {}
129     dg = {}
130
131     a[0], dg[0] = apply_activation(x, self.activators[0])
132     if dropout != None:
133         a[0] = dropout[0] * a[0]
134
135     for l in range(1, self.L + 1):
136         z = params[l].forward(a[l - 1])
137         a[l], dg[l] = apply_activation(z, self.activators[l])
138         if dropout != None:
139             a[l] = dropout[l] * a[l]
140
141     cache = {'a': a, 'dg': dg}
142     return cache
143
144 def cost_function(self, params, a, y, lambda_=0.01, eps=1e-8):
145     """
146     Parameters:
147     -----
148     params: Dict[LinearParameters]
149     a: array_like
150     y: array_like
151     lambda_: float
152             Default: 0.01
153     eps: float
154             Default: 1e-8
155
156     Returns:
157     -----
158     cost: float

```

```

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

```

```

206
207 def update_parameters(self, params, learning_rate=0.1):
208     """
209     Parameters:
210     -----
211     params : Dict[LinearParameters]
212             params[l].w = Weights
213             params[l].bias = Boolean
214             params[l].b = Bias
215     learning_rate : float
216             Default : 0.01
217
218     Returns:
219     -----
220     None
221     """
222     for param in params.values():
223         param.update(learning_rate)
224
225 def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000, print_cost=True):
226     """
227     Parameters:
228     -----
229     data : Dict[array_like]
230           data['x'] : array_like
231           data['y'] : array_like
232     learning_rate : float
233             Default : 0.1
234     lambda_ : float
235             Default : 0.0
236     num_iters : int
237             Default : 10000
238     print_cost_iter : int
239             Default: 1000    # 0 Doesn't print costs
240
241     Returns:
242     -----
243     costs : List[floats]
244     params : class[LinearParameters]
245     """
246     # Initialize parameters per layer
247     params = {}
248     for l in range(1, self.L + 1):
249         params[l] = LinearParameters(
250             (self.nodes[l], self.nodes[l - 1]), self.bias[l])
251
252     # Initialize RMSProp

```

```

253         rms_prop = RMSProp(params, self.bias)
254
255     # Initialize batching
256     batching = ShuffleBatchData(data, self.batch_size)
257
258     costs = []
259     for i in range(num_iters):
260         batches = batching.get_batches()
261         for batch in batches:
262             x = batch['x']
263             y = batch['y']
264             dropout = self.init_dropout(x.shape[1])
265             cache = self.forward_propagation(params, x, dropout)
266             cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
267             costs.append(cost)
268             self.backward_propagation(params, cache, y, dropout)
269             rms_prop.update(params, learning_rate)
270
271             if (print_cost_iter != 0) and (i % print_cost_iter == 0):
272                 print(f'Cost_after_iteration_{i}:_{cost}')
273
274     return params
275
276 def evaluate(self, params, x):
277     """
278     Parameters:
279     -----
280     params : Dict[LinearParameters]
281     x : array_like
282
283     Returns:
284     -----
285     y_hat : array_like
286     """
287     cache = self.forward_propagation(params, x)
288     a = cache['a'][self.L]
289     y_hat = (~(a < 0.5)).astype(int)
290     return y_hat
291
292 def accuracy(self, params, data):
293     """
294     Parameters:
295     -----
296     params : Dict[LinearParameters]
297     data : Dict[array_like]
298             data['x'] : array_like
299             data['y'] : array_like

```

```

300
301         Returns:
302         -----
303         accuracy : float
304         """
305         x = data['x']
306         y = data['y']

```

7.4 Adaptive Moment Estimation: The Adam Algorithm

We first note that with the momentum algorithm utilizing the EMA as it does, that it is an algorithm of the first moment (i.e., the mean of the gradients). Similarly, with RMSProp utilizing the square of the gradient as it does, we say it is an algorithm of the second moment (i.e., the uncentered variance of the gradients). Our goal is to utilize both gradient descent with momentum and RMSProp simultaneously to optimize our parameters. This combination of algorithms is called the *Adam algorithm* and is implemented as follows:

1. Initialize our parameters $w^{\{0\}}$ and $b^{\{0\}}$. Initialize $v_w^{\{0\}} = v_b^{\{0\}} = 0$ and $s_w^{\{0\}} = s_b^{\{0\}} = 0$. Fix our constants of momenta $\beta_1, \beta_2 \in [0, 1)$ and let $\epsilon > 0$ be sufficiently small.
2. For $0 \leq i < \text{num_iters}$:
 - a. Let $B = \lceil \frac{n}{b} \rceil$, and generate batches $\{\mathbb{X}^k\}$
 - b. For $1 \leq k \leq B$:
 - i. Apply forward propagation on \mathbb{X}^k .
 - ii. Compute the cost \mathbb{J} on \mathbb{X}^k .
 - iii. Apply backward propagation on \mathbb{X}^k to obtain

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

iv. Define

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

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

and define

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

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

v. Utilize bias correction via:

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

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

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

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

vi. Update the parameters:

$$w^{\{t\}} = w^{\{t-1\}} - \alpha \frac{\hat{v}_w^{\{t\}}}{\sqrt{\hat{s}_w^{\{t\}}} + \epsilon}$$

$$b^{\{t\}} = b^{\{t-1\}} - \alpha \frac{\hat{v}_b^{\{t\}}}{\sqrt{\hat{s}_b^{\{t\}}} + \epsilon}$$

We note that though we may still need to tune the hyper-parameter α , the hyper-parameters β_1, β_2 and ϵ typically work quite well with default values of

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

7.4.1 Python Implementation via numpy

```

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

```

8 class Adam():
9     def __init__(self, params, bias, beta1=0.9, beta2=0.999, eps=1e-8):
10         """
11         Parameters:
12         -----
13         params : Dict[LinearParameters]
14             params[l].w : array_like
15             params[l].b : array_like
16         bias : List[Boolean]
17         beta1 : float
18             Default: 0.9
19         beta2 : float
20             Default: 0.999
21         eps : float
22             Default: 10^{-8}
23
24         Returns:
25         None
26         """
27         self.beta1 = beta1
28         self.beta2 = beta2
29         self.eps = eps
30         self.bias = bias
31
32         self.mom = Momentum(params, self.bias, self.beta1)
33         self.rms_prop = RMSProp(params, self.bias, self.beta2, self.eps)
34
35     def update(self, params, iter, learning_rate=0.01):
36         """
37         Parameters:
38         -----
39         params : Dict[LinearParameters]
40             params[l].dw : array_like
41             params[l].db : array_like
42         iter : int
43         learning_rate : float
44             Default: 0.01
45
46         Returns:
47         None
48         """
49         self.mom.update(params, learning_rate, False)
50         self.rms_prop.update(params, learning_rate, False)
51
52         # Utilize bias correction
53         for l, param in params.items():
54             vw = self.mom.w[l] / (1 - self.beta1 ** iter)

```

```

55         sw = self.rms_prop.w[l] / (1 - self.beta2 ** iter)
56         param.w = param.w - learning_rate * (vw / (np.sqrt(sw) + self.eps))
57         if self.bias[l]:
58             vb = self.mom.b[l] / (1 - self.beta1 ** iter)
59             sb = self.rms_prop.b[l] / (1 - self.beta2 ** iter)
60             param.b = param.b - learning_rate * (vb / (np.sqrt(sb) + self.eps))
61
62     class NeuralNetwork():
63         def __init__(self, config):
64             """
65             Parameters:
66             -----
67             config : Dict
68                 config['lp_reg'] = 0,1,2
69                 config['batch_size'] = 2 ** p # p in {5, 6, 7, 8, 9, 10}
70                 config['nodes'] = List[int]
71                 config['bias'] = List[Boolean]
72                 config['activators'] = List[str]
73                 config['keep_probs'] = List[float]
74
75             Returns:
76             -----
77             None
78             """
79             self.config = config
80             self.lp_reg = config['lp_reg']
81             self.batch_size = config['batch_size']
82             self.nodes = config['nodes']
83             self.bias = config['bias']
84             self.activators = config['activators']
85             self.keep_probs = config['keep_probs']
86             self.L = len(config['nodes']) - 1
87
88         def init_dropout(self, num_examples, seed=101011):
89             """
90             Parameters:
91             -----
92             num_examples : int
93             seed : int
94                 Default: 1 # For reproducibility
95
96             Returns:
97             -----
98             D : Dict[layer : array_like]
99             """
100             np.random.seed(seed)
101             D = {}

```

```

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

```

```

149         params: Dict[LinearParameters]
150         a: array_like
151         y: array_like
152         lambda_: float
153             Default: 0.01
154         eps: float
155             Default: 1e-8
156
157         Returns:
158         -----
159         cost: float
160         """
161         n = y.shape[1]
162         if self.lp_reg == 0:
163             lambda_ = 0.0
164
165         # Compute regularization term
166         R = 0
167         for param in params.values():
168             R += np.sum(np.abs(param.w) ** self.lp_reg)
169         R *= (lambda_ / (2 * n))
170
171         # Compute unregularized cost
172         a = np.clip(a, eps, 1 - eps) # Bound a for stability
173         J = (-1 / n) * (np.sum(y * np.log(a) + (1 - y) * np.log(1 - a)))
174
175         cost = float(np.squeeze(J + R))
176
177         return cost
178
179     def backward_propagation(self, params, cache, y, dropout):
180         """
181         Parameters:
182         -----
183         params : Dict[LinearParameters]
184             params[l].w = Weights
185             params[l].bias = Boolean
186             params[l].b = Bias
187         cache : Dict[array_like]
188             cache['a'] : array_like
189             cache['dg'] : array_like
190         y : array_like
191
192         Returns:
193         -----
194         None
195         """

```

```

196
197     # Retrieve cache
198     a = cache['a']
199     dg = cache['dg']
200
201     # Initialize differentials along the network
202     delta = {}
203     delta[self.L] = ((a[self.L] - y) / y.shape[1]) * dropout[self.L]
204
205     for l in reversed(range(1, self.L + 1)):
206         delta[l - 1] = dg[l - 1] * \
207             params[l].backward(delta[l], a[l - 1]) * dropout[l - 1]
208
209 def update_parameters(self, params, learning_rate=0.1):
210     """
211     Parameters:
212     -----
213     params : Dict[LinearParameters]
214             params[l].w = Weights
215             params[l].bias = Boolean
216             params[l].b = Bias
217     learning_rate : float
218             Default : 0.01
219
220     Returns:
221     -----
222     None
223     """
224     for param in params.values():
225         param.update(learning_rate)
226
227 def fit(self, data, learning_rate=0.1, lambda_=0.01, num_iters=10000, print_cost=
228     """
229     Parameters:
230     -----
231     data : Dict[array_like]
232           data['x'] : array_like
233           data['y'] : array_like
234     learning_rate : float
235           Default : 0.1
236     lambda_ : float
237           Default : 0.0
238     num_iters : int
239           Default : 10000
240     print_cost_iter : int
241           Default: 1000    # 0 Doesn't print costs
242

```

```

243     Returns:
244     -----
245     costs : List[floats]
246     params : class[LinearParameters]
247     """
248     # Initialize parameters per layer
249     params = {}
250     for l in range(1, self.L + 1):
251         params[l] = LinearParameters(
252             (self.nodes[l], self.nodes[l - 1]), self.bias[l])
253
254     # Initialize Adam
255     adam = Adam(params, self.bias)
256
257     # Initialize batching
258     batching = ShuffleBatchData(data, self.batch_size)
259
260     costs = []
261     for i in range(num_iters):
262         batches = batching.get_batches()
263         B = len(batches)
264         for k in range(B):
265             t = i * B + k + 1
266             x = batches[k]['x']
267             y = batches[k]['y']
268             dropout = self.init_dropout(x.shape[1])
269             cache = self.forward_propagation(params, x, dropout)
270             cost = self.cost_function(
271                 params, cache['a'][self.L], y, lambda_)
272             costs.append(cost)
273             self.backward_propagation(params, cache, y, dropout)
274             adam.update(params, t, learning_rate)
275
276             if (print_cost_iter != 0) and (i % print_cost_iter == 0):
277                 print(f'Cost_after_iteration_{i}:_{cost}')
278
279     return params
280
281 def evaluate(self, params, x):
282     """
283     Parameters:
284     -----
285     params : Dict[LinearParameters]
286     x : array_like
287
288     Returns:
289     -----

```

```

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

```

7.5 Learning Rate Decay

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

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

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

7.6 Python Implementation

```
1 import copy
2
3 import numpy as np
4 from sklearn.utils import shuffle
5
6 import mllib.utils as utils
7
8 def get_batches(x, y, b):
9     """
10     Parameters
11     -----
12     x : array_like
13         x.shape = (m, n)
14     y : array_like
15         y.shape = (k, n)
16     b : int
17
18     Returns
19     -----
20     batches : List[Dict]
21         batches[i]['x'] : array_like
22             x.shape = (m, b) # except last batch
23             y.shape = (k, b) # except last batch
24
25     """
26     m, n = x.shape
27     ## Shuffle the data
28     x, y = shuffle(x.T, y.T) # Only shuffles rows, so transpose is needed
29     x = x.T
30     y = y.T
31
32     B = int(np.ceil(n / b))
33     batches = []
34     for i in range(B):
35         x_temp = x[:,(b * i):(b * (i + 1))]
36         y_temp = y[:,(b * i):(b * (i + 1))]
37         batches.append({'x' : x_temp, 'y' : y_temp})
38     # Slicing automatically ends at the end of
39     # the list if the stop is outside the index
40     return batches
41
42 def initialize_momenta(layers):
43     """
44     Parameters
45     -----
```

```

46     layers : List[int]
47         layers[l] = # nodes in layer l
48 Returns
49 -----
50 v : Dict[Dict[array_like]]
51 s : Dict[Dict[array_like]]
52 """
53 vw = {}
54 vb = {}
55 sw = {}
56 sb = {}
57 for l in range(1, len(layers)):
58     vw[l] = np.zeros((layers[l], layers[l - 1]))
59     sw[l] = np.zeros((layers[l], layers[l - 1]))
60     vb[l] = np.zeros((layers[l], 1))
61     sb[l] = np.zeros((layers[l], 1))
62
63 v = {'w' : vw, 'b' : vb}
64 s = {'w' : sw, 'b' : sb}
65
66 return v, s
67
68 def learning_rate_decay(epoch, learning_rate=0.01, decay_rate=0.0):
69     """
70     Parameters
71     -----
72     epoch : int
73     learning_rate : float
74         Default: 0.01
75     decay_rate : float
76         Default: 0.0 - Returns a constant learning_rate
77
78     Returns
79     -----
80     learning_rate : float
81     """
82     learning_rate = (1 / (1 + epoch * decay_rate)) * learning_rate
83     return learning_rate
84
85 def corrected_momentum(v, grads, update_iter, beta1=0.0):
86     """
87     Parameters
88     -----
89     v : Dict[Dict[array_like]]
90         v['w'][l].shape = w[l].shape
91         v['b'][l].shape = b[l].shape
92     grads : Dict[Dict]

```

```

93         grads['w'][l] : array_like
94         dw[l].shape = w[l].shape
95         grads['b'][l] : array_like
96         db[l].shape = b[l].shape
97     update_iter : int
98     beta1 : float
99         Default: 0.0 - Returns grads
100        Usual: 0.9
101
102     Returns
103     -----
104     v : Dict[Dict[array_like]]
105         v['w'][l].shape = dw[l].shape
106         v['b'][l].shape = db[l].shape
107     """
108     ## Retrieve velocities and gradients
109     vw = v['w']
110     vb = v['b']
111     dw = grads['w']
112     db = grads['b']
113     L = len(dw)
114
115     for l in range(1, L + 1):
116         vw[l] = beta1 * vw[l] + (1 - beta1) * dw[l]
117         vw[l] /= (1 - beta1 ** update_iter)
118         assert(vw[l].shape == dw[l].shape)
119         vb[l] = beta1 * vb[l] + (1 - beta1) * db[l]
120         vb[l] /= (1 - beta1 ** update_iter)
121         assert(vb[l].shape == db[l].shape)
122
123     v = {'w' : vw, 'b' : vb}
124     return v
125
126 def corrected_rmsprop(s, grads, update_iter, beta2=0.999):
127     """
128     Parameters
129     -----
130     s : Dict[Dict[array_like]]
131         s['w'][l].shape = w[l].shape
132         s['b'][l].shape = b[l].shape
133     grads : Dict[Dict]
134         grads['w'][l] : array_like
135         dw[l].shape = w[l].shape
136         grads['b'][l] : array_like
137         db[l].shape = b[l].shape
138     update_iter : int
139     beta2 : float

```

```

140         Default: 0.999
141
142     Returns
143     -----
144     s : Dict[Dict[array_like]]
145         s['w'][l].shape = w[l].shape
146         s['b'][l].shape = b[l].shape
147     """
148     ## Retrieve accelerations and gradients
149     sw = s['w']
150     sb = s['b']
151     dw = grads['w']
152     db = grads['b']
153     L = len(dw)
154
155     for l in range(1, L + 1):
156         sw[l] = beta2 * sw[l] + (1 - beta2) * (dw[l] * dw[l])
157         sw[l] /= (1 - beta2 ** update_iter)
158         assert(sw[l].shape == dw[l].shape)
159         sb[l] = beta2 * sb[l] + (1 - beta2) * (db[l] * db[l])
160         sb[l] /= (1 - beta2 ** update_iter)
161         assert(sb[l].shape == db[l].shape)
162
163     s = {'w' : sw, 'b' : sb}
164     return s
165
166
167 def update_parameters_adam(params, grads, epoch, batch_iter, v, s, momenta=[1e-8, 0.
168     """
169     Parameters
170     -----
171     params : Dict[Dict]
172         params['w'][l] : array_like
173             w[l].shape = (layers[l], layers[l-1])
174         params['b'][l] : array_like
175             b[l].shape = (layers[l], 1)
176     grads : Dict[Dict]
177         grads['dw'][l] : array_like
178             dw[l].shape = w[l].shape
179         grads['db'][l] : array_like
180             db[l].shape = b[l].shape
181     epoch : int
182     batch_iter : int
183     learning_rate : float
184         Default: 0.01
185     momenta : List[float]
186         momenta[0] = epsilon

```

```

187         Default: 10^{-8}
188     momenta[1] = beta_1
189         Default: 0.9
190     momenta[2] = beta_2
191         Default: 0.999
192
193     Returns
194     -----
195     params : Dict[Dict]
196         params['w'][1] : array_like
197             w[1].shape = (layers[1], layers[1-1])
198         params['b'][1] : array_like
199             b[1].shape = (layers[1], 1)
200     """
201     update_iter = epoch + batch_iter
202     ## Retrieve parameters
203     w = copy.deepcopy(params['w'])
204     b = copy.deepcopy(params['b'])
205     L = len(w)
206
207     ## Update velocities and accelerations
208     v = corrected_momentum(v, grads, update_iter, momenta[1])
209     vw = v['w']
210     vb = v['b']
211     s = corrected_rmsprop(s, grads, update_iter, momenta[2])
212     sw = s['w']
213     sb = s['b']
214
215     ## Update learning rate
216     learning_rate = learning_rate_decay(epoch, alpha0, decay_rate)
217
218     ## Perform update
219     for l in range(1, L + 1):
220         w[l] = w[l] - learning_rate * vw[l] / (np.sqrt(sw[l]) + momenta[0])
221         b[l] = b[l] - learning_rate * vb[l] / (np.sqrt(sb[l]) + momenta[0])
222
223     params = {'w' : w, 'b' : b}
224     return params
225
226 def model(x, y,
227         hidden_layer_sizes,
228         activators,
229         batch_size,
230         lambda_=0.0,
231         num_iters=10000,
232         print_cost=False):
233     """

```

```

234 Parameters
235 -----
236 x : array_like
237     x.shape = (layers[0], n)
238 y : array_like
239     y.shape = (layers[-1], n)
240 hidden_layer_sizes : List[int]
241     The number nodes layer l = hidden_layer_sizes[l-1]
242 activators : List[str]
243     activators[l] = activation function of layer l+1
244 batch_size : int
245 lambda_ : float
246     The regularization parameter
247     Default: 0.0
248 num_iters : int
249     Number of iterations with which our model performs gradient descent
250     Default: 10000
251 print_cost : Boolean
252     If True, print the cost every 1000 iterations
253     Default: False
254
255 Returns
256 -----
257 params : Dict[Dict]
258     params['w'][l] : array_like
259         w[l].shape = (layers[l], layers[l-1])
260     params['b'][l] : array_like
261         b[l].shape = (layers[l], 1)
262 cost : float
263     The final cost value for the optimized parameters returned
264 """
265 n, layers = utils.dim_retrieval(x, y, hidden_layer_sizes)
266 params = utils.initialize_parameters_random(layers)
267 v, s = initialize_momenta(layers)
268
269
270 ## main descent loop
271 for i in range(num_iters):
272     batches = get_batches(x, y, batch_size)
273     ## batch loop
274     batch_iter = 1
275     cost = 0
276     for batch in batches:
277         x = batch['x']
278         y = batch['y']
279         cache = utils.forward_propagation(x, params, activators)
280         cost += utils.compute_cost(y, params, cache)

```

```

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

```

8 Tuning Hyper-Parameters

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

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

Furthermore, suppose we impose a neural network architecture which has a collection of hyper-parameters (reabeled as):

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

The naive method of hyper-parameter tuning would instinctively be something of the form: Let $[d_i, d_i + k_i \Delta_i]$ denote an interval for which we require

$$\eta_i \in [d_i, d_i + k_i \Delta_i],$$

with an even-partition of

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

of length Δ_i . This collection forms a “grid” in \mathbb{R}^K for which each point of the grid gives us a full collection of hyper-parameters which we can then use to train our model. However, if certain hyper-parameters do not affect our model’s accuracy very much, we’ve added at least a full dimension of validation which is not needed. A more randomized approach would be best to determine such a hyper-parameter characterization must faster. Thus a random collection of points H_i for which we constrain $\eta_i \in H_i$.

How should we implement this set H_i ? Suppose for example, we wish to find

$$\eta_i \in [0.0001, 1],$$

but the majority of the random points will likely be in $[0.1, 1]$. Suppose we partition the interval

$$\begin{aligned} [0.0001, 1] &= 0.0001 < 0.001 < 0.01 < 0.1 < 1 \\ &= 10^{-4} < 10^{-3} < 10^{-2} < 10^{-1} < 10^0. \end{aligned}$$

This suggests we obtain a distribution of points using a logarithmic (in base 10) scale. Indeed, let

$$p \in [0, 1],$$

be a random point. Then letting $r = -4p \in [-4, 0]$, we obtain another random point, and let

$$H_i = \{10^{-4p} : p \in \text{rand}([0, 1])\},$$

for some prescribed set-cardinality. This allows us to choose more appropriately scaled-options for our hyper-parameters.

Remark 8.1. *Suppose we're using exponentially moving averages and have a hyper-parameter $\beta_1 \in [0, 1)$. If we do not use a log-scale, then the sensitivity of our model with respect to β_1 when $\beta_1 \approx 1$ is very strong. Indeed, we recall that when $\beta_1 = 0.999$, this corresponds to averaging over the previous 1000 days. And if we change β_1 slightly to*

$$\beta_1 = 0.9995,$$

then we've changed the interpretation of our model to the previous 2000 days. A subtle change for β_1 , but a drastic change to our model. The log-scale fixes this issue immediately.

We finally note that our hyper-parameters can become *stale* over time. That is, suppose we've trained a neural network, and tuned the hyper-parameters to allow an acceptable accuracy for our model. As the model refines over time, with more data being inserted to train on, it's import to re-test our hyper-parameters to make sure our model hasn't opened up to a better choice of one (or some or all) of the hyper-parameters we've previously tuned.

8.1 Python Implementation

```

1 def hyperparameter_scale(k, p):
2     """
3     Parameters
4     -----
5     k : int
6         The number random points to generate
7     p : int
8         The smallest magnitude for our log-scale
9
10    Returns
11    -----
12    hypers : List[float]
13        The list of hyper-parameters with which to tune
14    """
15    hypers = []
16    for _ in range(k):
17        r = p * np.random.rand()
18        hypers.append(10 ** r)
19    return hypers

```

9 Batch Normalization

See [7].

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

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

denote the mean and variance of the random-vector representation X of x , respectively. Then we consider the map

$$x_j \mapsto \frac{x_j - \mu}{\sigma} =: \hat{x}_j,$$

to be the *normalization* of x_j .

This definition is so “vanilla”, that it should be clear that this can be easily applied to each hidden-layer (we shall not use it on the output layer) of a neural network as well. However, we first note that there is an ambiguous choice amongst the implementation, namely, do we normalize $z^{[\ell]}$ or $a^{[\ell]}$, i.e., does normalization occur before or after we compute the activation unit. It seems more common to apply normalization to $z^{[\ell]}$, so that is what we do here without further mention of this choice.

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

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

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

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

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

normalized x , and consider

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

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

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

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

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

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

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

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

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

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

9.1 Backward Propagation

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

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

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

where

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

and

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

Define the compositional function

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

given by

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

This leads to compute some auxiliary differentials before continuing further.

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

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

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

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

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

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

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

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

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

Then the differentials

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

are given by

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

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

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

For fixed z , define the variance

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

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

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

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

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

are given by

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

Proof: Immediate from direct calculation. \square

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

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

with

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

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

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

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

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

are given by the diagonal matrices

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

Proof: We compute directly after noting that

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

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

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

as desired. \square

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

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

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

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

denote

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

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

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

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

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

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

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

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

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

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

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

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

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

Proof: Follows immediately from the previous Corollary. □

We now return to considering the compositional function

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

given by

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

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

•

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

and hence

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

•

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

yielding

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

•

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

and so

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

•

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

thus

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

•

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

and hence

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

Finally, since

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

we've described our desired gradients after summation.

9.2 Inferencing

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

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

and the variances

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

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

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

from which we average again to obtain

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

and the collection

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

from which we use the unbiased estimate

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

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

9.3 Algorithm Outline

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

1. Set hyper-parameters. Initialize parameters.

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

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

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

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

•

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

• For $\ell \in \{1, \dots, L-1\}$:

—

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

—

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

—

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

—

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

—

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

—

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

•

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

•

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

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

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

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

iv. Update parameters.

3. Compute

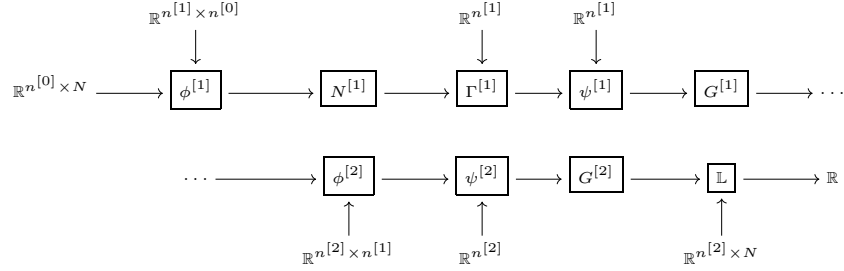
$$\begin{aligned} \mu^{[\ell]} &= \mathbb{E}[\mu^{[\ell]}_k], \\ \sigma^{2[\ell]} &= \frac{n}{n-1} \mathbb{E}[\sigma^{2[\ell]}_k] \end{aligned}$$

4. Return

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

9.4 Better Backpropagation

We consider a neural network utilizing batch normalization of the form



where we have the functions

1.

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

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

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

2.

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

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

3.

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

is given by

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

4.

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

is given by

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

where

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

5.

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

is the normalization operator given by

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

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

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

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

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

6.

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

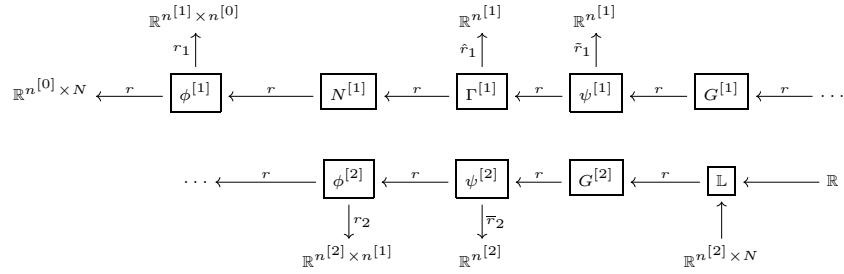
is given by

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

where

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

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



We consider our individual derivatives:

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

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

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

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

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

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

Then we have two differential paths to consider:

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

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

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

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

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

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

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

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

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

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

where

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

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

(a) In the network direction, we have that

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

with reverse differential

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

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

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

with reverse differential

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

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

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

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

(a) In the network direction, we have that

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

with reverse differential

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

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

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

with reverse differential

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

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

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

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

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,

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

This leads us to

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

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

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

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

and finally,

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

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

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

where C is the number of labels in our classification.

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

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

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

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

which clearly admits a bijection

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

Thus, we've effectively mapped our true labels

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

where

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

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

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

then we can declare a prediction via

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

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

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

and

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

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

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

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

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

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

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

where

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

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

To minimize our cost, we first note

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

then we see that

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

and

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

Hence,

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

yielding a gradient of

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

and similarly

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

and so

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

Finally, we conclude that

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

and

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

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

Part III

Convolutional Neural Networks

11 An Introduction to Convolutions

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

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

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

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

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

11.1 Cross-Correlation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 11.1. *Suppose*

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

and

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

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

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

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

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

$$(F * x)_1^1 = 1 * 1 + 0 * 2 + 1 * 4 + 1 * 5 = 10$$

$$(F * x)_2^1 = 1 * 2 + 0 * 0 + 1 * 5 + 1 * 6 = 13$$

$$(F * x)_3^1 = 1 * 0 + 0 * 3 + 1 * 6 + 1 * 0 = 6$$

$$(F * x)_1^2 = 1 * 4 + 0 * 5 + 1 * 0 + 1 * 1 = 5$$

$$(F * x)_2^2 = 1 * 5 + 0 * 6 + 1 * 1 + 2 * 2 = 10$$

$$(F * x)_3^2 = 1 * 6 + 0 * 0 + 1 * 2 + 1 * 3 = 11,$$

and hence

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

Example 11.2. Suppose

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

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

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

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

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

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

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

11.2 Convolution with Padding

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

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

Example 11.3. *Suppose*

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

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

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

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

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

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

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

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

and we write

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

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

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

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

11.3 Strided Convolution

We note that in our definition of a convolution

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

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

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

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

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

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

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

Example 11.4. *Suppose*

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

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

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

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

and hence that

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

or rather

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

11.4 Strided Convolutions with Padding

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

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

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

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

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

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

and now we immediately see by chasing the definitions that

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

Example 11.5. *Suppose*

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

and we have a filter

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

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

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

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

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

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

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

or rather

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

11.5 Convolutions Over Volumes

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

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

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

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

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

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

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

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

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

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

11.6 Multiple Filters

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

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

Then we have that

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

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

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

12 Convolutional Networks

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

12.1 Convolutional Layers (**conv**)

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

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

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

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

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

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

Letting

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

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

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

where

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

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

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

plus the bias terms

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

that is,

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

12.2 Pooling Layers (**pool**)

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

12.2.1 Max Pooling

Suppose

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

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

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

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

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

12.2.2 Average Pooling

Suppose

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

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

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

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

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

12.3 A Convolutional Network

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

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

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

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

This yields a network architecture of the following form:

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

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

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

12.4 Backpropagation

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

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

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

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

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

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

is given by

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

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

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

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

and hence

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

Next we consider

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

and hence

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

Finally, we consider

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

and hence

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

Appendices

A `utils.py`

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

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

B activators.py

```

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

```

```

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

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

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

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

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

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

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

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

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

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

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

It then follows that

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

reverseDifferential

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

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

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

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

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

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

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

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

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

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

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

and hence that

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

Proposition C.5. Suppose we have the compositional diagram

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

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

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

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

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

as desired. □

The following needs to be refined further still.

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

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

and in coordinates, we have that

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

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

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

It then follows that

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

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

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

evaluates to

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

and hence that

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

Lemma C.7. Suppose $f : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}^k$, and for $P \in \mathbb{R}^{n \times m}$, let $R = rf_P$. Then $R \in \mathbb{R}^k_n{}^m$ 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 $\frac{\partial}{\partial X_\mu^\nu} \in T_P \mathbb{R}^{n \times m}$ and $\frac{\partial}{\partial x^i} \in T_{f(P)} \mathbb{R}^k$ we have that

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

as desired. □

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.