Neural Networks

Matt R

June 28, 2022

Contents

Part I

Neural Networks and Deep Learning

1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have we have training examples $x \in \mathbb{R}^{n\times N}$ with binary labels $y \in \{0,1\}^{1\times N}$. We desire to train a model which yields an output a which represents
$$a = \mathbb{P}(y = 1|x).$$
To this end, let $\sigma : \mathbb{R} \to (0,1)$ denote the sigmoid function, i.e.,
$$\sigma(z) = \frac{1}{1+e^{-z}},$$
and let $w \in \mathbb{R}^{1\times n}$, $b \in \mathbb{R}$, and let
$$a = \sigma(wx + b).$$
To analyze the accuracy of model, we need a way to compare y and a, and ideally this functional comparison can be optimized with respect to (w, b) in such a way to minimize an error. To this end, we note that
$$\mathbb{P}(y|x) = a^y(1-a)^{1-y},$$
or rather
$$\mathbb{P}(y = 1|x) = a, \qquad \mathbb{P}(y = 0|x) = 1 - a,$$
so $\mathbb{P}(y|x)$ represents the *corrected probability*. Now since we want
$$a \approx 1 \quad \text{when } y = 1,$$
and
$$a \approx 0 \quad \text{when } y = 0,$$
and $0 \le a \le 1$, any error using differences won't be refined enough to analyze when tuning the model. Moreover, since introducing the sigmoid function, our usual mean-squared-error function won't be convex. This leads us to apply the log function, which when restricted to $(0,1)$ is a bijective mapping of $(0,1) \to (-\infty, 0)$. This leads us to define our log-loss function
$$\begin{aligned}\mathbb{L}(a, y) &= -\log(\mathbb{P}(y|x))\\ &= -\log\left(a^y(1-a)^{1-y}\right)\\ &= -\left[y\log(a) + (1-y)\log(1-a)\right],\end{aligned}$$

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

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

1.1 The Gradient

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

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

and

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

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

$$
\begin{tikzcd}
 & \mathbb{R}^{1\times n} \arrow[d, "w"'] & \mathbb{R} \arrow[d, "b"'] & & \{0,1\} \arrow[d, "y"'] & \\
\mathbb{R}^n \arrow[r, "x"] & \boxed{\phi} \arrow[r, "u"] & \boxed{\psi} \arrow[r, "z"] & \boxed{\sigma} \arrow[r, "a"] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R}
\end{tikzcd}
$$

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

$$
\begin{tikzcd}
 & \mathbb{R}^{1\times n} & \mathbb{R} & & \{0,1\} & \\
\mathbb{R}^n & \boxed{\phi} \arrow[l] \arrow[u, "r_1"] & \boxed{\psi} \arrow[l, "r"] \arrow[u, "\bar{r}_1"] & \boxed{\sigma} \arrow[l, "r"] & \boxed{\mathbb{L}} \arrow[l, "r"] \arrow[u] & \mathbb{R} \arrow[l]
\end{tikzcd}
$$

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

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

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

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

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

and hence that

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

2.
$$\psi : \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \qquad z := \psi(b,u) = u + b.$$

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

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

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

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

We then immediately have that

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

and

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

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

Then

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

4.

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

Then

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

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

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

and

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

1.2 Implementation in Python via **numpy**

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

```python
#! python3

import numpy as np

from mlLib.utils import apply_activation

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

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

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

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

```

```
        return z

    def backward(self, dz, x):
        """
        Parameters:
        -----------
        dz : array_like
        x : array_like

        Returns:
        --------
        None
        """
        if self.bias:
            self.db = np.sum(dz, axis=1, keepdims=True)
            assert (self.db.shape == self.b.shape)

        self.dw = np.einsum('ij,kj', dz, x)
        assert (self.dw.shape == self.w.shape)

    def update(self, learning_rate=0.01):
        """
        Parameters:
        -----------
        learning_rate : float
            Default : 0.01

        Returns:
        --------
        None
        """
        w = self.w - learning_rate * self.dw
        self.w = w

        if self.bias:
            b = self.b - learning_rate * self.db
            self.b = b

class LogisticRegression():
    def __init__(self, lp_reg):
        """
        Parameters:
        lp_reg : int
            2 : L_2 Regularization is imposed
            1 : L_1 Regularization is imposed
            0 : No regulariation is imposed

```

```python
        Returns:
        --------
        None
        """
        self.lp_reg = lp_reg

    def predict(self, params, x):
        """
        Parameters:
        -----------
        params : class[LinearParameters]
        x : array_like

        Returns:
        --------
        a : array_like
        dg : array_like
        """
        z = params.forward(x)
        a, dg = apply_activation(z, 'sigmoid')
        return a, dg

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

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

        R = np.sum(np.abs(params.w) ** self.lp_reg)
        R *= (lambda_ / (2 * n))

        a, _ = self.predict(params, x)
        a = np.clip(a, eps, 1 - eps)

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

```

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

        return cost

    def fit(self, x, y, learning_rate=0.1, lambda_=0.01, seed=1, num_iters=10000):
        """
        Parameters:
        -----------
        x : array_like
        y : array_like
        learning_rate : float
            Default : 0.1
        lambda_ : float
            Default : 0.0
        num_iters : int
            Default : 10000

        Returns:
        --------
        costs : List[floats]
        params : class[Parameters]
        """
        dims = (y.shape[0], x.shape[0])
        n = x.shape[1]
        params = LinearParameters(dims, True, seed)

        if self.lp_reg == 0:
            lambda_ = 0.0

        costs = []
        for i in range(num_iters):
            a, _ = self.predict(params, x)
            cost = self.cost_function(params, x, y, lambda_)
            costs.append(cost)
            dz = (a - y) / n
            params.backward(dz, x)
            params.update(learning_rate)

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

        return params

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

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

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

        return y_hat

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

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

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

        return accuracy
```

1.3 Implementation in Python via sklearn

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

```
#! python3

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

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

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

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

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

2 Neural Networks: A Single Hidden Layer

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

$$
\begin{tikzcd}
& \mathbb{R}^{1\times n^{[0]}} \arrow[d, "w"'] & \mathbb{R} \arrow[d, "b"'] & & \{0,1\} \arrow[d, "y"'] & \\
\mathbb{R}^{n^{[0]}} \arrow[r, "x"] & \boxed{\phi} \arrow[r, "u"] & \boxed{\psi} \arrow[r, "z"] & \boxed{\sigma} \arrow[r, "a"] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R}
\end{tikzcd}
$$

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:

$$
\begin{tikzcd}
& \mathbb{R}^{n^{[1]}\times n^{[0]}} \arrow[d, "w^{[1]}"'] & \mathbb{R}^{n^{[1]}} \arrow[d, "b^{[1]}"'] & & \mathbb{R}^{1\times n^{[1]}} \arrow[d, "w^{[2]}"'] & \mathbb{R} \arrow[d, "b^{[2]}"'] & & \{0,1\} \arrow[d, "y"'] & \\
\mathbb{R}^{n^{[0]}} \arrow[r, "a^{[0]}:=x"] & \boxed{\phi^{[1]}} \arrow[r, "u^{[1]}"] & \boxed{\psi^{[1]}} \arrow[r, "z^{[1]}"] & \boxed{G^{[1]}} \arrow[r, "a^{[1]}"] & \boxed{\phi^{[2]}} \arrow[r, "u^{[2]}"] & \boxed{\psi^{[2]}} \arrow[r, "z^{[2]}"] & \boxed{G^{[2]}} \arrow[r, "a^{[2]}"] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R}
\end{tikzcd}
$$

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

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

- $$\psi : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n, \qquad z := \psi(b, u) = u + b,$$

- $$G : \mathbb{R}^n \to \mathbb{R}^n, \qquad a := G(z),$$

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

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

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

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

castingDifferential

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

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

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

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

We use the notation

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

and thus may write

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

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

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

Proof: We calculate

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

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

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

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

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

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

and the result follows. $\square$

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

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

We note that since

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

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

2.1.2 The Hyperbolic Tangent Function

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

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

We then calculate

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

Furthermore, we note that

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

Indeed,

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

as desired.

2.1.3 The Rectified Linear Unit Function

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

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

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

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

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

We then calculate

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

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 $\mathrm{softmax}(z)$ given by

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

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

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

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

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

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

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

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

2.2 Backward Propagation

We consider a neural network of the form

$$
\begin{tikzcd}
 & \mathbb{R}^{n^{[1]}\times n^{[0]}} \arrow[d, "w^{[1]}"'] & \mathbb{R}^{n^{[1]}} \arrow[d, "b^{[1]}"'] & & \mathbb{R}^{n^{[2]}\times n^{[1]}} \arrow[d, "w^{[2]}"'] & \mathbb{R}^{n^{[2]}} \arrow[d, "b^{[2]}"'] & & \mathbb{R}^{n^{[2]}} \arrow[d, "y"'] & \\
\mathbb{R}^{n^{[0]}} \arrow[r, "a^{[0]}:=x"] & \boxed{\phi^{[1]}} \arrow[r, "u^{[1]}"] & \boxed{\psi^{[1]}} \arrow[r, "z^{[1]}"] & \boxed{G^{[1]}} \arrow[r, "a^{[1]}"] & \boxed{\phi^{[2]}} \arrow[r, "u^{[2]}"] & \boxed{\psi^{[2]}} \arrow[r, "z^{[2]}"] & \boxed{G^{[2]}} \arrow[r, "a^{[2]}"] & \boxed{\mathbb{L}} \arrow[r] & \mathbb{R}
\end{tikzcd}
$$

where we have the functions:

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

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

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

   is given by

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

3. $$\psi^{[\ell]} : \mathbb{R}^{n^{[\ell]}} \times \mathbb{R}^{n^{[\ell]}} \to \mathbb{R}^{n^{[\ell]}}$$

   is given by

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

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

   is the given loss-function.

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

$$\begin{tikzcd}
 & \mathbb{R}^{n^{[1]} \times n^{[0]}} & \mathbb{R}^{n^{[1]}} & & \mathbb{R}^{n^{[2]} \times n^{[1]}} & \mathbb{R}^{n^{[2]}} & & \mathbb{R}^{n^{[2]}} \arrow[d] & \\
\mathbb{R}^{n^{[0]}} & \boxed{\phi^{[1]}} \arrow[l, "r"'] \arrow[u, "r_1"] & \boxed{\psi^{[1]}} \arrow[l, "r"'] \arrow[u, "\overline{r}_1"] & \boxed{G^{[1]}} \arrow[l, "r"'] & \boxed{\phi^{[2]}} \arrow[l, "r"'] \arrow[u, "r_2"] & \boxed{\psi^{[2]}} \arrow[l, "r"'] \arrow[u, "\overline{r}_2"] & \boxed{G^{[2]}} \arrow[l, "r"'] & \boxed{\mathbb{L}} \arrow[l, "r"'] & \mathbb{R} \arrow[l]
\end{tikzcd}$$

First, we need to consider our individual derivatives:

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

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

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

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

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

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

   Then we have two differentials to consider:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Include the following two results in the Reverse Differential appendix once created.

Proposition 2.3. *Suppose we have the compositional diagram*

$$\mathbb{R}^n \xrightarrow{f} \mathbb{R}^m \xrightarrow{g} \mathbb{R}^k \xrightarrow{h} \mathbb{R}^l$$

and we let $F = h \circ g \circ f : \mathbb{R}^n \to \mathbb{R}^l$. Then for any $x \in \mathbb{R}^n$ and any $\zeta \in T_{F(x)}\mathbb{R}^l$, the reverse derivative satisfies

$$rF_x(\zeta) = rf_x \circ rg_{f(x)} \circ rh_{g(f(x))}(\zeta).$$

Proof: For any $\xi \in T_x\mathbb{R}^n$ and any $\zeta \in T_{F(x)}\mathbb{R}^l$, we have by definition

$$\begin{aligned}
\langle rF_x(\zeta), \xi \rangle_{\mathbb{R}^n} &= \langle \zeta, dF_x(\xi) \rangle_{\mathbb{R}^l} \\
&= \left\langle \zeta, dh_{g(f(x))} \circ dg_{f(x)} \circ df_x(\xi) \right\rangle_{\mathbb{R}^l} \\
&= \left\langle rh_{g(f(x))}(\zeta), dg_{f(x)} \circ df_x(\xi) \right\rangle_{\mathbb{R}^k} \\
&= \left\langle rg_{f(x)} \circ rh_{g(f(x))}(\zeta), df_x(\xi) \right\rangle_{\mathbb{R}^m} \\
&= \left\langle rf_x \circ rg_{f(x)} \circ rh_{g(f(x))}(\zeta), \xi \right\rangle_{\mathbb{R}^n}
\end{aligned}$$

as desired. □

Lemma 2.4. *Suppose $f : \mathbb{R}^{n\times m} \to \mathbb{R}^k$, and for $P \in \mathbb{R}^{n\times m}$, let $R = rf_P$. Then $R \in \mathbb{R}^k{}_n{}^m$ is rank $(1,2)$-tensor written in coordinates as*

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

and the components is given by

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

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

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

as desired. $\square$

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

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

and we have our cost function

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

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

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

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

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

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

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

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

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

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

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

and hence

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

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

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

and hence

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

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

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

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

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

and hence

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

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

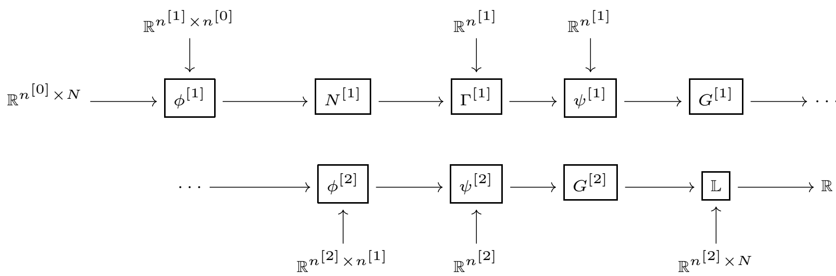
$$\begin{aligned}
r_1F_p &= r_1(\mathbb{L} \circ G^{[2]} \circ \psi^{[2]} \circ \phi^{[2]} \circ G^{[1]} \circ \psi^{[1]} \circ \phi^{[1]})_p \\
&= r_1\phi^{[1]}_p \circ r\psi^{[1]}_{u^{[1]}} \circ rG^{[1]}_{z^{[1]}} \circ r\phi^{[2]}_{a^{[1]}} \circ r\psi^{[2]}_{u^{[2]}} \circ rG^{[2]}_{z^{[2]}} \circ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

and hence

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

3 Deep Neural Networks

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



3.1 Backward Propagation

As the general derivation for backpropagation can be easily (if not tediously) generalized from **??** using induction, we give the general outline for computational purposes.

Let $\mathbb{L} : \mathbb{R}^{m_L} \times \mathbb{R}^{m_L} \to \mathbb{R}$ be a generic loss function, and suppose our cost function is given by the usual

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

Then from previous computations, we have the following gradients for any $\ell \in \{1, 2, ..., L\}$, that

$$\frac{\partial \mathbb{J}}{\partial W^{[\ell]}} = \frac{1}{n} \delta^{[\ell]} a^{[\ell-1]T}$$
$$\frac{\partial \mathbb{J}}{\partial b^{[\ell]}} = \frac{1}{n} \sum_{j=1}^{n} \delta^{[\ell]}{}_j$$

where we impose the notation of

$$a^{[0]} := x.$$

So we need only give a full characterization of $\delta^{[\ell]}$.. To this end, we define recursively starting at layer-L by

$$\begin{aligned}
\delta^{[L]T} &:= d(\mathbb{L}_y)_{a^{[L]}} \cdot dg^{[L]}_{z^{[L]}}, \\
\delta^{[L-1]T} &:= \delta^{[L]T} \cdot W^{[L]} \cdot dg^{[L-1]}_{z^{[L-1]}}, \\
&\vdots \\
\delta^{[\ell]T} &:= \delta^{[\ell+1]T} W^{[\ell+1]} dg^{[\ell]}_{z^{[\ell]}}, \\
&\vdots \\
\delta^{[1]T} &:= \delta^{[2]T} W^{[2]} dg^{[1]}_{z^{[1]}},
\end{aligned}$$

as desired.

3.2 Implementation in Python via numpy

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

```
#! python3

import numpy as np

from mlLib.utils import LinearParameters, apply_activation

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

        Returns:
        --------
        None
        """
        self.config = config
        self.lp_reg = config['lp_reg']
        self.nodes = config['nodes']
```

```python
        self.bias = config['bias']
        self.activators = config['activators']
        self.L = len(config['nodes']) - 1

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

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

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

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

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

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

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

        Returns:
```

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

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

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

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

        return cost

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

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

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

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

```

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

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

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

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

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

```
            cost = self.cost_function(params, cache['a'][self.L], y, lambda_)
            costs.append(cost)
            self.backward_propagation(params, cache, y)
            self.update_parameters(params, learning_rate)

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

        return params

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

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

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

        Returns:
        --------
        accuracy : float
        """
        y_hat = self.evaluate(params, x)
        acc = np.sum(y_hat == y) / y.shape[1]

        return acc
```

3.3 Implementation in Python via `tensorflow`

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

```python
#! python3

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow import keras
from keras import Model, Input
from keras.layers import Dense

def keras_functional_nn(csv):
    df = pd.read_csv(csv)
    dataset = df.values
    x, y = dataset[:, :-1], dataset[:, -1].reshape(-1, 1)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.15)
    train = {'x' : x_train, 'y' : y_train}
    test = {'x' : x_test, 'y' : y_test}
    mu = np.mean(train['x'], axis=0, keepdims=True)
    var = np.var(train['x'], axis=0, keepdims=True)
    train['x'] = (train['x'] - mu) / np.sqrt(var)
    test['x'] = (test['x'] - mu) / np.sqrt(var)

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

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

    ## Compile desired model
    model.compile(
```

```
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )

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

    ## Evaluate the model
    test_scores = model.evaluate(test['x'], test['y'], verbose=2)
    print(f'Test Loss: {test_scores[0]}')
    print(f'Test Accuracy: {test_scores[1]}')
```

References