Neural Networks

Matt R

March 1, 2022

Contents

1 Logistic Regression

We begin with a review of binary classification and logistic regression. To this end, suppose we have we have training examples $x \in \mathbb{R}^{m \times n}$ 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 $w \in \mathbb{R}^m$, $b \in \mathbb{R}$, and let

$$a = \sigma(w^T x + b).$$

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 the error. To this end, we note that

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

or rather

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

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

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

and

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

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

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

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

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

1.1 The Gradient

To compute the gradient of our cost function $\mathbb{J}$, we first write $\mathbb{J}$ as a sum of compositions as follows: We have the log-loss function considered as a map $\mathbb{L} : (0,1) \times \mathbb{R} \to \mathbb{R}$,

$$\mathbb{L}(a, y) = -\left[y \log(a) + (1-y)\log(1-a)\right],$$

we have the sigmoid function $\sigma : \mathbb{R} \to (0,1)$ with $\sigma(z) = a$ and $\sigma'(z) = a(1-a)$, and we have the collection of affine-functionals $\phi_x : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ given by

$$\phi_x(w, b) = w^T x + b,$$

for which we fix an arbitrary $x \in \mathbb{R}^m$ and write $\phi = \phi_x$, and set $z = \phi(w, b)$. Finally, we introduce the auxiliary function $\mathcal{L} : \mathbb{R}^m \times \mathbb{R} \to \mathbb{R}$ given by

$$\mathcal{L}(w, b) = \mathbb{L}(\sigma(\phi(w, b)), y).$$

Then by the chain rule, we have that

$$\begin{aligned}
d\mathcal{L} &= d_a\mathbb{L}(a, y) \circ d\sigma(z) \circ d_w\phi(w, b) \\
&= \left[-\frac{y}{a} + \frac{1-y}{1-a}\right] \cdot a(1-a) \cdot \begin{bmatrix} x^T & 1 \end{bmatrix} \\
&= \left[-y(1-a) + a(1-y)\right] \cdot \begin{bmatrix} x^T & 1 \end{bmatrix} \\
&= (a - y) \begin{bmatrix} x^T & 1 \end{bmatrix}
\end{aligned}$$

Composition turns into matrix multiplication in the tangent space.

Moreover, since in Euclidean space, we have that $\nabla f = (df)^T$, and hence that

$$\nabla \mathcal{L}(w, b) = (a - y) \begin{bmatrix} x \\ 1 \end{bmatrix},$$

or rather

$$\partial_w \mathbb{L}(a, y) = (a - y)x, \qquad \partial_b \mathbb{L}(a, y) = a - y.$$

Finally, since our cost function $\mathbb{J}$ is the sum-log-loss, we have by linearity that

$$\begin{aligned} \partial_w \mathbb{J}(w, b) &= \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j) x_j \\ &= \frac{1}{n} ((a - y) \cdot x^T)^T \\ &= \frac{1}{n} x \cdot (a - y)^T \end{aligned}$$

and

$$\partial_b \mathbb{J}(w, b) = \frac{1}{n} \sum_{j=1}^{n} (a_j - y_j).$$

1.1.1 Vectorization in Python

Here we include the general code to train a model using logistic regression without regularization and without tuning on a cross-validation set.

```python
import copy

import numpy as np

def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
    """

    sigma = (1 / (1 + np.exp(-z)))
    return sigma

```

```
def cost_function(x, y, w, b):
    """
    Parameters
    -----------
    x : array_like
        x.shape = (m, n) with m-features and n-examples
    y : array_like
        y.shape = (1, n)
    w : array_like
        w.shape = (m, 1)
    b : float

    Returns
    -------
    J : float
        The value of the cost function evaluated at (w, b)
    dw : array_like
        dw.shape = w.shape = (m, 1)
        The gradient of J with respect to w
    db : float
        The partial derivative of J with respect to b
    """

    # Auxiliary assignments
    m, n = x.shape
    z = w.T @ x + b
    assert z.size == n
    a = sigmoid(z).reshape(1, n)
    dz = a - y

    # Compute cost J
    J = (-1 / n) * (np.log(a) @ y.T + np.log(1 - a) @ (1 - y).T)

    # Compute dw and db
    dw = (x @ dz.T) / m
    assert dw.shape == w.shape
    db = np.sum(dz) / m

    return J, dw, db

def grad_descent(x, y, w, b, alpha=0.001, num_iters=2000, print_cost=False):
    """
    Parameters
    ----------
    x, y, w, b : See cost_function above for specifics.
        w and b are chosen to initialize the descent (likely all components 0)
    alpha : float
```

```
        The learning rate of gradient descent
    num_iters : int
        The number of times we wish to perform gradient descent

    Returns
    -------
    costs : List[float]
        For each iteration we record the cost-values associated to (w, b)
    params : Dict[w : array_like, b : float]
        w : array_like
            Optimized weight parameter w after iterating through grad descent
        b : float
            Optimized bias parameter b after iterating through grad descent
    grads : Dict[dw : array_like, db : float]
        dw : array_like
            The optimized gradient with repsect to w
        db : float
            The optimized derivative with respect to b
    """

    costs = []
    w = copy.deepcopy(w)
    b = copy.deepcopy(b)
    for i in range(num_iters):
        J, dw, db = cost_function(x, y, w, b)
        w = w - alpha * dw
        b = b - alpha * db

        if i % 100 == 0:
            costs.append(J)
            if print_cost:
                idx = int(i / 100) - 1
                print(f'Cost after iteration {i}: {costs[idx]}')

    params = {'w' : w, 'b' : b}
    grads = {'dw' : dw, 'db' : db}

    return costs, params, grads

def predict(w, b, x):
    """
    Parameters
    ----------
    w : array_like
        w.shape = (m, 1)
    b : float
    x : array_like
```

```
        x.shape = (m, n)

    Returns
    -------
    y_predict : array_like
        y_pred.shape = (1, n)
        An array containing the prediction of our model applied to training
        data x, i.e., y_pred = 1 or y_pred = 0.
    """

    m, n = x.shape
    # Get probability array
    a = sigmoid(w.T @ x + b)
    # Get boolean array with False given by a < 0.5
    pseudo_predict = ~(a < 0.5)
    # Convert to binary to get predictions
    y_predict = pseudo_predict.astype(int)

    return y_predict

def model(x_train, y_train, x_test, y_test, alpha=0.001, num_iters=2000, accuracy=Tr
    """
    Parameters:
    -----------
    x_train, y_train, x_test, y_test : array_like
        x_train.shape = (m, n_train)
        y_train.shape = (1, n_train)
        x_test.shape = (m, n_test)
        y_test.shape = (1, n_test)
    alpha : float
        The learning rate for gradient descent
    num_iters : int
        The number of times we wish to perform gradient descent
    accuracy : Boolean
        Use True to print the accuracy of the model

    Returns:
    d : Dict
        d['costs'] : array_like
            The costs evaluated every 100 iterations
        d['y_train_preds'] : array_like
            Predicted values on the training set
        d['y_test_preds'] : array_like
            Predicted values on the test set
        d['w'] : array_like
            Optimized parameter w
        d['b'] : float
```

```
            Optimized parameter b
        d['learning_rate'] : float
            The learning rate alpha
        d['num_iters'] : int
            The number of iterations with which gradient descent was performed

    """

    m = x_train.shape[0]
    # initialize parameters
    w = np.zeros((m, 1))
    b = 0.0
    # optimize parameters
    costs, params, grads = grad_descent(x_train, y_train, w, b, alpha, num_iters)
    w = params['w']
    b = params['b']
    # record predictions
    y_train_preds = predict(w, b, x_train)
    y_test_preds = predict(w, b, x_test)
    # group results into dictionary for return
    d = {'costs' : costs,
         'y_train_preds' : y_train_preds,
         'y_test_preds' : y_test_preds,
         'w' : w,
         'b' : b,
         'learning_rate' : alpha,
         'num_iters' : num_iters}

    if accuracy:
        train_acc = 100 - np.mean(np.abs(y_train_preds - y_train)) * 100
        test_acc = 100 - np.mean(np.abs(y_test_preds - y_test)) * 100
        print(f'Training Accuracy: {train_acc}%')
        print(f'Test Accuracy: {test_acc}%')


    return d
```

2 Neural Networks: A Single Hidden Layer

Suppose we wish to consider the binary classification problem given the training set (x, y) with $x \in \mathbb{R}^{s_0 \times n}$ and $y \in \{0, 1\}^{1 \times n}$. Usually with logistic regression we have the following type of structure:

$$[x^1, ..., x^{s_0}] \xrightarrow{\varphi} [z] \xrightarrow{g} [a] \xrightarrow{=} \hat{y},$$

where

$$z = \varphi(x) = w^T x + b,$$

is our affine-linear transformation, and

$$a = g(z) = \sigma(z)$$

is our sigmoid function. 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), 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:

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \underbrace{\xrightarrow{\varphi^{[1]}} \begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

where

$$\varphi^{[1]} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_1}, \qquad \varphi^{[1]}(x) = W^{[1]}x + b^{[1]},$$
$$\varphi^{[2]} : \mathbb{R}^{s_1} \to \mathbb{R}, \qquad \varphi^{[2]}(x) = W^{[2]}x + b^{[2]},$$

and $W^{[1]} \in \mathbb{R}^{s_1 \times s_0}, W^{[2]} \in \mathbb{R}^{1 \times s_1}, b^{[1]} \in \mathbb{R}^{s_1}, b^{[2]} \in \mathbb{R}$, and $g^{[\ell]}$ is a *broadcasted* activator function (e.g., the sigmoid function $\sigma(z)$, or $\tanh(z)$, or $\mathrm{ReLU}(z)$). Such a network is called a 2-layer neural network where x is the input layer (called layer-0), $a^{[1]}$ is a hidden layer (called layer-1), and $a^{[2]}$ is the output layer (called layer-2).

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

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

where $v \in \mathbb{R}^m$ and $\{e_i : 1 \leq i \leq m\}$ is the standard basis for $\mathbb{R}^m$. In practice, we will 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 $G : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of g from $\mathbb{R}$ to $\mathbb{R}^m$. Then the differential $dG_z : T_z\mathbb{R}^m \to T_{G(z)}\mathbb{R}^m$ is given by*

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

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

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

Proof: We calculate

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

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. $\square$

Let us lay out all of these functions explicitly (in the Smooth Category) as to facilitate our later computations for our cost function and our gradients. To this end:

$$\begin{aligned} &\varphi^{[1]} : \mathbb{R}^{s_0} \to \mathbb{R}^{s_1}, && d\varphi^{[1]} : T\mathbb{R}^{s_0} \to T\mathbb{R}^{s_1}, \\ &z^{[1]} = \varphi^{[1]}(x) = W^{[1]}x + b^{[1]}, && d\varphi^{[1]}_x(v) = W^{[1]}v; \end{aligned}$$

$$g^{[1]} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_1}, \qquad dg^{[1]} : T\mathbb{R}^{s_1} \to T\mathbb{R}^{s_1},$$
$$a^{[1]} = g^{[1]}(z^{[1]}), \qquad \frac{\partial a^{[1]\mu}}{\partial z^{[1]\nu}} = \delta^\mu_\nu g^{[1]\prime}(z^{[1]\mu});$$

$$\varphi^{[2]} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_2}, \qquad d\varphi^{[2]} : T\mathbb{R}^{s_1} \to T\mathbb{R}^{s_2},$$
$$z^{[2]} = \varphi^{[2]}(a^{[1]}) = W^{[2]}a^{[1]} + b^{[2]}, \qquad d\varphi^{[2]}_{a^{[2]}}(v) = W^{[2]}v;$$

$$g^{[2]} : \mathbb{R}^{s_2} \to \mathbb{R}^{s_2}, \qquad dg^{[2]} : T\mathbb{R}^{s_2} \to T\mathbb{R}^{s_2},$$
$$a^{[2]} = g^{[2]}(z^{[2]}), \qquad \frac{\partial a^{[2]\mu}}{\partial z^{[2]\nu}} = \delta^\mu_\nu g^{[2]\prime}(z^{[2]\mu}).$$

That is, given an input $x \in \mathbb{R}^{s_0}$, we get a predicted value $\hat{y} \in \mathbb{R}^{s_2}$ of the form

$$\hat{y} = g^{[2]} \circ \varphi^{[2]} \circ g^{[1]} \circ \varphi^{[1]}(x).$$

This compositional function is known as *forward propagation*.

2.1 Backpropagation

Since we wish to optimize our model with respect to our parameter $W^{[\ell]}$ and $b^{[\ell]}$, we consider a generic loss function $\mathbb{L} : \mathbb{R}^{s_2} \times \mathbb{R}^{s_2} \to \mathbb{R}$, $\mathbb{L}(\hat{y}, y)$, and by acknowledging the potential abuse of notation, we assume y is fixed, and consider the aforementioned as a function of a single-variable

$$\mathbb{L}_y : \mathbb{R}^{s_2} \to \mathbb{R}, \qquad \mathbb{L}_y(\hat{y}) = \mathbb{L}(\hat{y}, y).$$

We also define the function

$$\Phi(A, u, \xi) = A\xi + u,$$

and note that we're suppressing a dependence on the layer ℓ which only affects our domain and range of Φ (and not the actual calculations involving the derivatives). Moreover, in coordinates we see that

$$\begin{aligned} \frac{\partial \Phi^i}{\partial A^\mu_\nu} &= \frac{\partial}{\partial A^\mu_\nu}(A^i_j \xi^j + u^i) \\ &= (\delta^i_\mu \delta^\nu_j \xi^j) \\ &= \delta^i_\mu \xi^\nu; \end{aligned}$$

$$
\begin{aligned}
\frac{\partial \Phi^i}{\partial u^\mu} &= \frac{\partial}{\partial u^\mu}(A^i_j \xi^j + u^i) \\
&= \delta^i_\mu;
\end{aligned}
$$

and

$$
\begin{aligned}
\frac{\partial \Phi^i}{\xi^\mu} &= \frac{\partial}{\partial \xi^\mu}(A^i_j \xi^j + u^i) \\
&= A^i_j \delta^j_\mu \\
&= A^i_\mu.
\end{aligned}
$$

We now define the compositional function

$$F : \mathbb{R}^{s_2 \times s_1} \times \mathbb{R}^{s_2} \times \mathbb{R}^{s_1 \times s_0} \times \mathbb{R}^{s_1} \times \mathbb{R}^{s_0} \to \mathbb{R}$$

given by

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1} \times \mathbb{1} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

We first introduce an error term $\delta^{[2]} \in \mathbb{R}^{s_2}$ defined by

$$
\begin{aligned}
\delta^{[2]} &:= \nabla(\mathbb{L}_y \circ g^{[2]})(z^{[2]}) \\
&= (d\mathbb{L}_y \circ g^{[2]})_{z^{[2]}})^T.
\end{aligned}
$$

Now we calculate the gradient $\frac{\partial F}{\partial C}$ in coordinates by $\delta^{[2]} = d_{z^{[2]}} F$

$$
\begin{aligned}
\frac{\partial F}{\partial C^\mu_\nu} &= \frac{\partial}{\partial C^\mu_\nu}\left[\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, a^{[1]})\right] \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \frac{\partial}{\partial C^\mu_\nu}(C^j_i a^{[1]i} + c^j) \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \delta^j_\mu a^{[1]\nu} \\
&= \delta^{[2]}{}_\mu a^{[1]\nu} \\
&= [a^{[1]} \delta^{[2]T}]^\nu_\mu
\end{aligned}
$$

and hence that

$$
\begin{aligned}
\frac{\partial F}{\partial C} &= \left[\frac{\partial F}{\partial C^\mu_\nu}\right]^T \\
&= \left[\delta^{[2]}_\mu a^{[1]\nu}\right]^T \\
&= \delta^{[2]} a^{[1]T}.
\end{aligned}
$$

Moreover, we also calculate

$$\frac{\partial F}{\partial c^{\mu}} = \sum_{j=1}^{s_2} \delta^{[2]j} \delta^{j}_{\mu},$$

and hence that

$$\frac{\partial F}{\partial c} = \delta^{[2]}.$$

Next we introduce another error term $\delta^{[1]} \in \mathbb{R}^{s_1}$ defined by

$$\delta^{[1]} = [dg^{[1]}_{z^{[1]}}]^T C^T \delta^{[2]}$$

with coordinates $\delta^{[1]} = d_{z^{[1]}}F$

$$\begin{aligned}
(\delta^{[1]\mu})^T &= \sum_{i=1}^{s_2} \sum_{j=1}^{s_1} \delta^{[2]i} C^{i}_{j} g^{[1]\prime}(z^{[1]j}) \delta^{j}_{\mu} \\
&= \sum_{i=1}^{s_2} \delta^{[2]i} C^{i}_{\mu} g^{[1]\prime}(z^{[1]\mu})
\end{aligned}$$

and now calculate the gradient $\frac{\partial F}{\partial B}$ in coordinates by

$$\begin{aligned}
\frac{\partial F}{\partial B^{\mu}_{\nu}} &= \frac{\partial}{B^{\mu}_{\nu}} \left[\mathbb{L}_y \circ g^{[2]} \circ \Phi(C, c, g^{[1]}(Bx + b)) \right] \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^{\rho}} \sum_{\lambda=1}^{s_1} \frac{\partial a^{[1]\rho}}{\partial z^{[1]\lambda}} \frac{\partial \Phi^{\lambda}}{\partial B^{\mu}_{\nu}} \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^{\rho}} \sum_{\lambda=1}^{s_1} \delta^{\rho}_{\lambda} g^{[1]\prime}(z^{[1]\rho}) \delta^{\lambda}_{\mu} x^{\nu} \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} \frac{\partial \Phi^j}{\partial \xi^{\rho}} \delta^{\rho}_{\mu} g^{[1]\prime}(z^{[1]\rho}) x^{\nu} \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} \sum_{\rho=1}^{s_1} C^{j}_{\rho} \delta^{\rho}_{\mu} g^{[1]\prime}(z^{[1]\rho}) x^{\nu} \\
&= \sum_{j=1}^{s_2} \delta^{[2]j} C^{j}_{\mu} g^{[1]\prime}(z^{[1]\mu}) x^{\nu} \\
&= \delta^{[1]}{}_{\mu} x^{\nu} \\
&= \left[x \delta^{[1]T} \right]^{\nu}_{\mu},
\end{aligned}$$

and hence that

$$\frac{\partial F}{\partial B} = \left[\frac{\partial F}{\partial B^{\mu}_{\nu}}\right]^T$$
$$= \delta^{[2]} x^T.$$

Moreover, from the above calculation, we immediately see that

$$\frac{\partial F}{\partial b^{\mu}} = \delta^{[1]}.$$

In summary, we've computed the following gradients

$$\frac{\partial F}{\partial W^{[2]}} = \delta^{[2]} a^{[1]T}$$
$$\frac{\partial F}{\partial b^{[2]}} = \delta^{[2]}$$
$$\frac{\partial F}{\partial W^{[1]}} = \delta^{[1]} x^T$$
$$\frac{\partial F}{\partial b^{[1]}} = \delta^{[1]},$$

where

$$\delta^{[2]} = [d(\mathbb{L}_y \circ g^{[2]})_{z^{[2]}}]^T$$
$$\delta^{[1]} = [dg^{[1]}_{z^{[1]}}]^T C^T \delta^{[2]}.$$

Finally, we recall that our cost function $\mathbb{J}$ is the average sum of our loss function $\mathbb{L}$ over our training set, we get that

$$\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) = \frac{1}{n}\sum_{j=1}^{n} F(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}, x_j),$$

and hence that

$$\frac{\partial \mathbb{J}}{\partial W^{[2]}} = \frac{1}{n}\sum_{j=1}^{n} \delta^{[2]}{}_j a^{[1]}{}_j{}^T = \frac{1}{n}\delta^{[2]} a^{[1]T}$$

$$\frac{\partial \mathbb{J}}{\partial b^{[2]}} = \frac{1}{n}\sum_{j=1}^{n} \delta^{[2]}{}_j$$

$$\frac{\partial \mathbb{J}}{\partial W^{[1]}} = \frac{1}{n}\sum_{j=1}^{n} \delta^{[1]}{}_j x_j^T = \frac{1}{n}\delta^{[1]} x^T$$

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

2.2 Activation Functions

There are mainly only a handful of activating functions we consider for our non-linearity conditions.

2.2.1 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

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

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

Moreover, suppose that $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of σ from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\sigma(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned} dg_z(v) &= \left.\frac{d}{dt}\right|_{t=0} g(z+tv) \\ &= \left.\frac{d}{dt}\right|_{t=0} (\sigma(z^i + tv^i)) \\ &= (\sigma'(z^i)v^i) \\ &= (\sigma(z^i)(1-\sigma(z^i))v^i) \\ &= g(z) \odot (1 - g(z)) \odot v, \end{aligned}$$

where $\odot$ represents the Hadamard product (or component-wise multiplication); or rather, as as a matrix in $\mathbb{R}^{m\times m}$,

$$[dg_z]^\mu_\nu = \delta^\mu_\nu \sigma(z^\mu)(1-\sigma(z^\mu)).$$

2.2.2 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

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

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of tanh from $\mathbb{R}$ to $\mathbb{R}^m$, then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = (\tanh(z^i)),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned}dg_z(v) &= [\tanh'(z^i)] \odot [v^i] \\ &= [1 - \tanh^2(z^i)] \odot [v^i] \\ &= \delta^i_j(1 - \tanh^2(z^i))v^j.\end{aligned}$$

2.2.3 The Rectified Linear 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.

Suppose $g : \mathbb{R}^m \to \mathbb{R}^m$ is the broadcasting of ReLU from $\mathbb{R}$ to $\mathbb{R}^m$. Then for $z = (z^1, ..., z^m) \in \mathbb{R}^m$, we have that

$$g(z) = \mathrm{ReLU}(z^i; \beta),$$

and $dg_z : T_z\mathbb{R}^m \to T_{g(z)}\mathbb{R}^m$ given by

$$\begin{aligned} dg_z(v) &= [\mathrm{ReLU}'(z^i; \beta)] \odot [v^i] \\ &= \delta^i_j(\beta\chi_{(-\infty,0)}(z^i) + \chi_{[0,\infty)}(z^i))v^j. \end{aligned}$$

2.2.4 The Softmax Function

We finally have the softmax function 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 on our outer-layer to obtain a probability distribution over our predicted labels. We then calculate for $z = (z^1, ..., z^m) \in \mathbb{R}^m$ that $d(\mathrm{softmax})_z : T_z\mathbb{R}^m \to T_{\mathrm{softmax}(z)}\mathbb{R}^m$

$$\begin{aligned} d(\mathrm{softmax})_z(v) &= \left.\frac{d}{dt}\right|_{t=0} \mathrm{softmax}(z + tv) \\ &= \left.\frac{d}{dt}\right|_{t=0} \frac{1}{\sum_{j=1}^m e^{z^j + tv^j}} \begin{pmatrix} e^{z^1+tv^1} \\ e^{z^2+tv^2} \\ \vdots \\ e^{z^m+tv^m} \end{pmatrix} \\ &= \frac{-1}{\left(\sum_{j=1}^m e^{z^j}\right)^2} \left(\sum_{j=1}^m e^{z^j} v^j\right) \begin{pmatrix} e^{z^1} \\ \vdots \\ e^{z^m} \end{pmatrix} + \frac{1}{\sum_{j=1}^m e^{z^j}} \begin{pmatrix} e^{z^1}v^1 \\ \vdots \\ e^{z^m}v^m \end{pmatrix} \\ &= -\langle \mathrm{softmax}(z), v\rangle \,\mathrm{softmax}(z) + \mathrm{softmax}(z) \odot v, \end{aligned}$$

or rather in coordinates

$$[d(\mathrm{softmax})_z]^i_j = S^i(\delta^i_j + \delta_{\rho j} S^\rho),$$

where

$$S^\mu = x^\mu \circ \mathrm{softmax}(z).$$

2.3 Binary Classification - An Example

We return the network given by

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{=} \hat{y},$$

and show how such a model would be trained using python below. We assume layer-2 has the sigmoid function (since it's binary classification) as an activator and our hidden layer has the ReLU function as activators.

We note that $s_2 = 1$ since we're dealing with a single activator in this layer, and

$$a^{[2]} = g^{[2]}(z^{[2]}) = \sigma(z^{[2]}),$$

with

$$d(g^{[2]})_{z^{[2]}} = \sigma'(z^{[2]}) = \sigma(z^{[2]})(1 - \sigma(z^{[2]})) = a^{[2]}(1 - a^{[2]}).$$

In layer-1, we have that

$$a^{[1]} = g^{[1]}(z^{[1]}) = \mathrm{ReLU}(z^{[1]}),$$

with

$$d(g^{[1]})_{z^{[1]}} = \left[\delta^\mu_\nu \chi_{[0,\infty)}(z^{[1]\mu})\right]^\mu_\nu .$$

Finally, we choose our loss function $\mathbb{L}(\hat{y}, y)$ to be the log-loss function (since we're using the sigmoid activator on the outer-layer), i.e.,

$$\mathbb{L}(\hat{y}, y) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y}),$$

or rather

$$\mathbb{L}(x, y) = -y\log(a^{[2]}) - (1 - y)\log(1 - a^{[2]}).$$

We then have the cost function $\mathbb{J}$ given by

$$\begin{aligned}\mathbb{J}(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}) &= \frac{-1}{n}\sum_{j=1}^{n}\left(y_j \log({a^{[2]}}_j) + (1-y_j)\log(1-{a^{[2]}}_j)\right) \\ &= \frac{-1}{n}\left(\left\langle y, \log(a^{[2]})\right\rangle + \left\langle 1-y, \log(1-a^{[2]})\right\rangle\right)\end{aligned}$$

Moreover, when using backpropagation, we see that

$$\begin{aligned}{\delta^{[2]}}^T_j &= d(\mathbb{L}_{y_j})_{a^{[2]}} \cdot d(g^{[2]})_{{z^{[2]}}_j} \\ &= \left(-\frac{y_j}{{a^{[2]}}_j} + \frac{1-y_j}{1-{a^{[2]}}_j}\right)\cdot({a^{[2]}}_j(1-{a^{[2]}}_j) \\ &= {a^{[2]}}_j - y_j,\end{aligned}$$

or rather

$$\delta^{[2]} = a^{[2]} - y.$$

Similarly, we compute

$$\begin{aligned}{\delta^{[1]}}^T_j &= {\delta^{[2]}}^T_j W^{[2]}[dg^{[1]}_{{z^{[1]}}_j}] \\ &= {\delta^{[2]}}^T_j W^{[2]}[\delta^\mu_\nu \cdot \chi_{[0,\infty)}({z^{[1]}}^\mu_j)]\end{aligned}$$

2.3.1 Vectorization in Python

```
import copy

import numpy as np

# Activator functions

def sigmoid(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    -------
    sigma : array_like
        The value of the sigmoid function evaluated at z
    ds : array_like
        The differential of the sigmoid function evaluate at z
```

```
    """
    # Compute value of sigmoid
    sigma = (1 / (1 + np.exp(-z)))
    # Compute differential of sigmoid
    ds = sigma * (1 - sigma)
    return sigma, ds

# Preliminary functions for our model
def layer_shapes(x, y, hidden_layer_size):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m_x, n)
    y : array_like
        y.shape = (m_y, n)
    hidden_layer_size : int
        The number nodes in the hidden layer
    Returns
    -------
    n : int
        The number of training examples
    m_x : int
        The number of input features
    m_h : The number of nodes in the hidden layer
    m_y : The number of nodes in the output layer
    """
    m_x, n = x.shape
    assert(y.shape[1] == n)
    m_y = y.shape[0]
    m_h = hidden_layer_size
    return n, m_x, m_h, m_y



def initialize_parameters(m_x, m_h, m_y):
    """
    Parameters
    ----------
    m_x : int
        The number of input features
    m_h : int
        The number of nodes in the hidden layer
    m_y : int
        The number of nodes in the output layer

    Returns
```

```
    -------
    params : Dict
        w1 : array_like
            w1.shape = (m_h, m_x)
        b1 : array_like
            b1.shape = (m_h, 1)
        w2 : array_like
            w2.shape= (m_y, m_h)
        b2 : array_like
            b2.shape = (m_y, 1)
    """
    w1 = np.random.randn(m_h, m_x) * 0.01
    b1 = np.zeros((m_h, 1))
    w2 = np.random.randn(m_y, m_h) * 0.01
    b2 = np.zeros((m_y, 1))

    params = {'w1' : w1,
              'b1' : b1,
              'w2' : w2,
              'b2' : b2}

    return params

def forward_propagation(x, params):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m_x, n)
    params : Dict
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
            b1.shape = (m_h, 1)
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
    Returns
    -------
    a2 : array_like
        a2.shape = (m_y, n)
    cache : Dict
        cache['z1'] : array_like
            z1.shape = (m_h, n)
        cache['a1'] : array_like
            a1.shape = (m_h, n)
```

```
        cache['z2'] : array_like
            z2.shape = (m_y, n)
        cache['a2'] = a2
    """

    # Retrieve parameters
    w1 = params['w1']
    b1 = params['b1']
    w2 = params['w2']
    b2 = params['b2']

    # Auxiliary computations
    z1 = w1 @ x + b1
    a1 = np.tanh(z1)
    z2 = w2 @ a1 + b2
    a2 = sigmoid(z2)

    assert(a1.shape == (w1.shape[0], x.shape[1]))
    assert(a2.shape == (w2.shape[0], a1.shape[1]))

    cache = {'z1' : z1,
             'a1' : a1,
             'z2' : z2,
             'a2' : a2}

    return a2, cache

def compute_cost(a2, y):
    """
    Parameters
    ----------
    a2 : array_like
        a2.shape = (m_y, n)
    y : array_like
        y.shape = (m_y, n)
    Returns
    -------
    cost : float
        The cost evaluated at y and a2
    """
    n = y.shape[1]
    cost = (-1 / n) * (np.sum(y * np.log(a2)) + np.sum((1 - y) * np.log(1 - a2)))
    cost = float(np.squeeze(cost))  # Makes sure we return a float

    return cost

def backward_propagation(params, cache, x, y):
```

```
    """
    Parameters
    ----------
    params : Dict
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
            b1.shape = (m_h, 1)
    cache : Dict
        cache['z1'] : array_like
            z1.shape = (m_h, n)
        cache['a1'] : array_like
            a1.shape = (m_h, n)
        cache['z2'] : array_like
            z2.shape = (m_y, n)
        cache['a2'] = a2
    x : array_like
        x.shape = (m_x, n)
    y : array_like
        y.shape = (m_y, n)
    Returns
    -------
    grads : Dict
        grads['dw2'] : array_like
            dw2.shape = (m_y, m_h)
        grads['db2'] : array_like
            db2.shape = (m_y, 1)
        grads['dw1'] : array_like
            dw1.shape = (m_h, m_x)
        grads['db1'] : array_like
            db1.shape = (m_h, 1)
    """
    # Retrieve parameters
    w1 = params['w1']
    w2 = params['w2']

    # Set dimensional constants
    m_x, n = x.shape
    m_y, m_h = w2.shape

    # Retrieve node outputs
    a1 = cache['a1']
    a2 = cache['a2']
```

```python

    # Auxiliary Computations
    delta2 = a2 - y
    assert(delta2.shape ==(m_y, n))
    d_tanh = 1 - (a1 * a1)
    assert(d_tanh.shape == (m_h, n))
    delta1 = (w2.T @ delta1) * d_tanh
    assert(delta1.shape == (m_h, n))

    # Gradient computations
    dw2 = (1 / n) * delta2 @ a1.T
    db2 = (1 / n) * np.sum(delta2, axis=1, keepdims=True)
    dw1 = (1 / n) * delta1 @ x.T
    db1 = (1 / n) * np.sum(delta1, axis=1, keepdims=True)

    # Combine and return dict
    grads = {'dw2' : dw2,
             'db2' : db2,
             'dw1' : dw1,
             'db1' : db1}
    return grads

def update_parameters(params, grads, learning_rate=1.2):
    """
    Parameters
    ----------
    params : Dict
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
            b1.shape = (m_h, 1)
    grads : Dict
        grads['dw2'] : array_like
            dw2.shape = (m_y, m_h)
        grads['db2'] : array_like
            db2.shape = (m_y, 1)
        grads['dw1'] : array_like
            dw1.shape = (m_h, m_x)
        grads['db1'] : array_like
            db1.shape = (m_h, 1)
    learning_rate : float
        Default = 1.2
    Returns
```

```
    -------
    params : Dict
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
            b1.shape = (m_h, 1)
    """
    # Retrieve parameters
    w2 = copy.deepcopy(params['w2'])
    b2 = params['b2']
    w1 = copy.deepcopy(params['w1'])
    b1 = params['b1']

    # Retrieve gradients
    dw2 = grads['dw2']
    db2 = grads['db2']
    dw1 = grads['dw1']
    db1 = grads['db1']

    # Perform update
    w2 = w2 - learning_rate * dw2
    b2 = b2 - learning_rate * db2
    w1 = w1 - learning_rate * dw1
    b1 = b1 - learning_rate * db1

    # Combine and return dict
    params = {'w2' : w2,
              'b2' : b2,
              'w1' : w1,
              'b1' : b1}
    return params


# The main neural network training model
def model(x, y, num_hidden_layer, num_iters=10000, print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m_x, n)
    y : array_like
        y.shape = (m_y. n)
    num_hidden_layer : int
```

```
        Number of nodes in the single hidden layer
    num_iters : int
        Number of iterations with which our model performs gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations
    Returns
    -------
    params : Dict
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
            b1.shape = (m_h, 1)
    """
    # Set dimensional constants
    n, m_x, m_h, m_y = layer_shapes(x, y, num_hidden_layer)
    # initialize parameters
    params = initialize_parameters(m_x, m_h, m_y)

    # main loop for gradient descent
    for i in range(num_iters):
        a2, cache = forward_propagation(X, params)
        cost = compute_cost(a2, y)
        grads = backward_propagation(params, cache, x, y)
        params = update_parameters(params, grads)

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

    return params

# Using our model to obtain predictions
def predict(params, x):
    """
    Parameters
    ----------
    params : Dict
        params['w2'] : array_like
            w2.shape = (m_y, m_h)
        params['b2'] : array_like
            b2.shape = (m_y, 1)
        params['w1'] : array_like
            w1.shape = (m_h, m_x)
        params['b1'] : array_like
```

```
            b1.shape = (m_h, 1)
    x : array_like
        x.shape = (m_x, n)

    Returns
    -------
    predictions : array_like
        predictions.shape = (m_y, n)
    """
    a2, _ = forward_propagation(x, params)
    predictions = np.zeros(a2.shape)
    predictions[~(a2 < 0.5)] = 1

    return predictions
```

3 Deep Neural Networks

Sup