Neural Networks

Matt R

March 9, 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 $x \in \mathbb{R}^{m \times n}$ 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, for function $f : \mathbb{R}^N \to \mathbb{R}$ 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.

```
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,
          learning_rate=0.001,
          num_iters=2000, accuracy=False):
    """
    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)
    learning_rate : 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, learning_rate, num_i
    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' : learning_rate,
         '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
```

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 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:

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

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. □

Returning to our network, 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:

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

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

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

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

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.*

backPropDerivation

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

$$F(C, c, B, b, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi \circ (\mathbb{1}_{\mathbb{R}^{s_2 \times s_1}} \times \mathbb{1}_{\mathbb{R}^{s_2}} \times (g^{[1]} \circ \Phi))(C, c, B, b, x).$$

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

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

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

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

where

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

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

$$\begin{aligned}\frac{\partial \mathbb{J}}{\partial W^{[2]}} &= \frac{1}{n}\sum_{j=1}^{n} \delta^{[2]}{}_j a^{[1]}{}_j{}^T = \frac{1}{n}\delta^{[2]} a^{[1]T} \\ \frac{\partial \mathbb{J}}{\partial b^{[2]}} &= \frac{1}{n}\sum_{j=1}^{n} \delta^{[2]}{}_j \\ \frac{\partial \mathbb{J}}{\partial W^{[1]}} &= \frac{1}{n}\sum_{j=1}^{n} \delta^{[1]}{}_j x_j^T = \frac{1}{n}\delta^{[1]} x^T \\ \frac{\partial \mathbb{J}}{\partial b^{[1]}} &= \frac{1}{n}\sum_{j=1}^{n} \delta^{[1]}{}_j\end{aligned}$$

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 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.

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

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

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 Random Initialization

In the section that follows, we see that to begin gradient descent for a shallow neural network, we initialize our parameters $b^{[\ell]}$ to be 0, but choose an arbitrarily small, but nonzero initialization for $W^{[\ell]}$. Let's see why we choose $W^{[\ell]}$ to be nonzero. Indeed, suppose we initialize with $b^{[\ell]} = 0$ and $W^{[\ell]} = 0$. Then we see that

$$\delta^{[1]T} = \delta^{[2]}W^{[2]}dg^{[1]}_{z^{[1]}} = 0,$$

and so

$$\frac{\partial \mathbb{J}}{\partial W^{[1]}} = \frac{1}{n}\delta^{[1]}x^T = 0.$$

Then we conclude that our parameter $W^{[1]}$ remains at 0 during every iteration which is enough reason to not initialize $W^{[2]}$ at 0. Similarly, since

$$a^{[1]} = \tanh(W^{[1]}x + b^{[1]}) = \tanh(0) = 0,$$

we reach a similar conclusion about $W^{[1]}$ and $W^{[2]}$, respectively.

2.3.2 Vectorization in Python

```python
import copy

import numpy as np

import activators
from activators import ACTIVATORS

# Preliminary functions for our model
def dim_retrieval(x, y, hidden_sizes):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[L], n)
    hidden_sizes : List[int]
        hidden_sizes[i-1] = The number nodes layer i
    Returns
    -------
    n : int
        The number of training examples
    layers : List
        layer[l] = # nodes in layer l

    """
    m, n = x.shape
    assert(y.shape[1] == n)
    K = y.shape[0]
    layers = [m]
    layers.extend(hidden_sizes)
    layers.append(K)

    return n, layers

## Initialize parameters using the size of each layer
def initialize_parameters_random(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    params : Dict[Dict]
```

```
        w[l] : array_like
            dwl.shape = (layers[l], layers[l-1])
        b[l] : array_like
            dbl.shape = (layers[l], 1)
    """
    w = {}
    b = {}
    for l in range(1, len(layers)):
        w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
        b[l] = np.zeros((layers[l], 1))
    params = {'w' : w, 'b' : b}
    return params

def forward_propagation(x, params):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m_x, n)
    params : Dict[Dict]
        w[l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        b[l] : array_like
            b[l].shape = (layers[l], 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
    w = params['w']
    b = params['b']
    w1 = w[1]
    b1 = b[1]
    w2 = w[2]
    b2 = b[2]

    # Auxiliary computations
    z1 = w1 @ x + b1
    a1, _1 = activators.tanh(z1)
    z2 = w2 @ a1 + b2
    a2, _2 = activators.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[Dict]
        w[l] : array_like
            dwl.shape = (layers[l], layers[l-1])
        b[l] : array_like
            dbl.shape = (layers[l], 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
    w = params[’w’]
    w1 = w[1]
    w2 = w[2]

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

    # Retrieve node outputs
    a1 = cache[’a1’]
    a2 = cache[’a2’]

    # 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 @ delta2) * d_tanh
    assert(delta1.shape == (m_h, n))

    # Gradient computations
    dw = {}
    db = {}
    dw[2] = (1 / n) * delta2 @ a1.T
    db[2] = (1 / n) * np.sum(delta2, axis=1, keepdims=True)
    dw[1] = (1 / n) * delta1 @ x.T
    db[1] = (1 / n) * np.sum(delta1, axis=1, keepdims=True)

    # Combine and return dict
    grads = {'dw' : dw, 'db' : db}
    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
    w = copy.deepcopy(params['w'])
    b = params['b']

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

    # Perform update
    w[2] = w[2] - learning_rate * dw[2]
    b[2] = b[2] - learning_rate * db[2]
    w[1] = w[1] - learning_rate * dw[1]
    b[1] = b[1] - learning_rate * db[1]

    # Combine and return dict
    params = {'w' : w, 'b' : b}
    return params


# The main neural network training model
def model(x, y, hidden_sizes, num_iters=10000, print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m_x, n)
    y : array_like
        y.shape = (m_y. n)
    hidden_sizes : 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[Dict[array_like]]
        params['w'][2] : array_like
            w[2].shape = (m_y, m_h)
        params['b'][2] : array_like
            b[2].shape = (m_y, 1)
        params['w'][1] : array_like
            w[1].shape = (m_h, m_x)
        params['b'][1] : array_like
            b[1].shape = (m_h, 1)
    """
    # Set dimensional constants
    n, layers = dim_retrieval(x, y, hidden_sizes)
    # initialize parameters
    params = initialize_parameters_random(layers)

    # 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

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

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer } 0} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer } 1} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]s_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]s_2} \end{bmatrix}}_{\text{Layer } 2} \xrightarrow{\varphi^{[3]}} \cdots$$

$$\cdots \xrightarrow{\varphi^{[L-1]}} \underbrace{\begin{bmatrix} z^{[L-1]1} \\ \vdots \\ z^{[L-1]s_{L-1}} \end{bmatrix} \xrightarrow{g^{[L-1]}} \begin{bmatrix} a^{[L-1]1} \\ \vdots \\ a^{[L-1]s_{L-1}} \end{bmatrix}}_{\text{Layer } L-1} \xrightarrow{\varphi^{[L]}} \underbrace{\begin{bmatrix} z^{[L]1} \\ \vdots \\ z^{[L]s_L} \end{bmatrix} \xrightarrow{g^{[L]}} \begin{bmatrix} a^{[L]1} \\ \vdots \\ a^{[L]s_L} \end{bmatrix}}_{\text{Layer } L} \xrightarrow{=} \begin{bmatrix} \hat{y}^1 \\ \vdots \\ \hat{y}^{s_L} \end{bmatrix},$$

where

$$s_\ell := \text{ the number of nodes in layer-}\ell,$$

$$\varphi^{[\ell]} : \mathbb{R}^{s_{\ell-1}} \to \mathbb{R}^{s_\ell}, \qquad \varphi^{[\ell]}(\xi) = W^{[\ell]}\xi + b^{[\ell]}, \qquad W^{[\ell]} \in \mathbb{R}^{s_\ell \times s_{\ell-1}}, b \in \mathbb{R}^{s_\ell},$$

and

$$g^{[\ell]} : \mathbb{R}^{s_\ell} \to \mathbb{R}^{s_\ell},$$

is a broadcasted activation function determined by the layer-ℓ.

As with a shallow network, our functional composition to obtain $a^{[L]}$ is known as forward propagation.

3.1 Backpropagation

As the general derivation for backpropagation can be easily (if not tediously) generalized from Section 2.1 using induction, we give the general outline for computational purposes.

Let $\mathbb{L} : \mathbb{R}^{s_L} \times \mathbb{R}^{s_L} \to \mathbb{R}$ 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.

3.1.1 Vectorization in Python

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.

```
import copy

import numpy as np

import utils
import activators
from activators import ACTIVATORS


## Auxiliary functions for model composition
```

```
def initialize_parameters(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    params : Dict[Dict]
        w[l] : array_like
            dwl.shape = (layers[l], layers[l-1])
        b[l] : array_like
            dbl.shape = (layers[l], 1)
    """
    w = {}
    b = {}
    for l in range(1, len(layers)):
        w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
        b[l] = np.zeros((layers[l], 1))
    params = {’w’ : w, ’b’ : b}
    return params

## Compute activation unit
def linear_activation_forward(a_prev, w, b, activator):
    """
    Parameters
    ----------
    a_prev : array_like
        a_prev.shape = (layers[l], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    b : array_like
        b.shape = (layers[l+1], 1)
    activator : str
        activator = ’relu’, ’sigmoid’, or ’tanh’

    Returns
    -------
    z : array_like
        z.shape = (layer_dims[l+1], n)
    a : array_like
        a.shape = (layer_dims[l+1], n)
    """
    assert activator in ACTIVATORS, f’{activator}␣is␣not␣a␣valid␣activator.’
    z = w @ a_prev + b
    if activator == ’relu’:
        a, _ = activators.relu(z)
    elif activator == ’sigmoid’:
        a, _ = activators.sigmoid(z)
    elif activator == ’tanh’:
        a, _ = activators.tanh(z)

    assert(z.shape == a.shape)
    return z, a

def forward_propagation(x, params, activators):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params[’w’][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    Returns
    -------
    cache : Dict[Dict]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params[’w’]
    b = params[’b’]
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Initialize a
    a[0] = x
    for l in range(1, L + 1):
        z[l], a[l] = linear_activation_forward(a[l - 1], w[l], b[l], activators[l -

    cache = {’a’ : a, ’z’ : z}
    return cache
```

```

# Compute the cost
def compute_cost(y, cache):
    """
    Parameters
    ----------
    y : array_like
        y.shape = (layers[-1], n)
    cache : Dict[Dict]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)

    Returns
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
    a = cache[’a’]
    L = len(a)
    aL = a[L - 1]

    cost = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
    cost = float(np.squeeze(cost))

    return cost

def linear_activation_backward(delta_next, z, w, activator):
    """
    Parameters
    ----------
    delta_next : array_like
        delta_next.shape = (layers[l+1], n)
    z : array_like
        z.shape = (layers[l+1], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    activator : str
        activator = ’relu’, ’sigmoid’, or ’tanh’

    Returns
    -------
    delta : array_like
        delta.shape = (layers[l], n)
    """
    assert activator in ACTIVATORS, f’{activator}␣is␣not␣a␣valid␣activator.’

    n = delta_next.shape[1]

    if activator == ’relu’:
        _, dg = activators.relu(z)
    elif activator == ’sigmoid’:
        _, dg = activators.sigmoid(z)
    elif activator == ’tanh’:
        _, dg = activators.tanh(z)

    da = w.T @ delta_next
    assert(da.shape == (w.shape[1], n))
    delta = da * dg
    assert(delta.shape == (w.shape[1], n))
    return delta

def backward_propagation(x, y, params, cache, activators):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    Returns
    -------
    grads : Dict[Dict]
        grads[’dw’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’db’][l] : array_like
            db[l].shape = b[l].shape
    """
    ## Retrieve parameters
    a = cache[’a’]
    z = cache[’z’]
    w = params[’w’]
    n = x.shape[1]
    L = len(z)

    ## Compute deltas
    delta = {}
    delta[L] = a[L] - y
    for l in reversed(range(1, L)):
        delta[l] = linear_activation_backward(delta[l + 1], z[l], w[l + 1], activato

    ## Compute gradients
    dw = {}
    db = {}
    for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * delta[l] @ a[l - 1].T
        assert(dw[l].shape == w[l].shape)
    grads ={’dw’ : dw, ’db’ : db}
    return grads

def update_parameters(params, grads, learning_rate=0.01):
    """
    Parameters
    ----------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    grads : Dict[Dict]
        grads[’dw’][l] : array_like
            dw[l].shape = w[l].shape
        grads[’db’][l] : array_like
            db[l].shape = b[l].shape
    learning_rate : float
        Default: 0.01
        The learning rate for gradient descent

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
```

```
            b[l].shape = (layers[l], 1)
    """
    ## Retrieve parameters
    w = copy.deepcopy(params[’w’])
    b = copy.deepcopy(params[’b’])
    L = len(w)

    ## Retrieve gradients
    dw = grads[’dw’]
    db = grads[’db’]

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

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


## The main model for training our parameters
def model(x, y, hidden_layer_sizes, activators, num_iters=10000, print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[function]
        activators[l] = activation function of layer l+1
    num_iters : int
        Number of iterations with which our model performs gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations

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

    ## main loop
    for i in range(num_iters):
        cache = forward_propagation(x, params, activators)
        cost = compute_cost(cache, y)
        grads = backward_propagation(x, y, params, cache, activators)
        params = update_parameters(params, grads, 0.1)

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

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

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

where $\mathfrak{X}$ 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 following the following ratios:

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

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

and

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

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

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

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

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

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

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

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

- If $\mathcal{E}(\mathfrak{X}) < \epsilon$ and $\mathcal{E}(\mathfrak{X}) < \mathcal{E}(\mathcal{D}) <\sim 10\epsilon$, then we say our model has *high variance* since our model is overfitting the data.

- If $\mathcal{E}(\mathfrak{X}) \approx \mathcal{E}(\mathcal{D}) >\sim 10\epsilon$, then we say our model has *high bias* since our model is underfitting the data.

- If $10\epsilon \sim< \mathcal{E}(\mathfrak{X}) \ll \mathcal{E}(\mathcal{D})$, 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 $\mathcal{E}(\mathfrak{X}), \mathcal{E}(\mathcal{D}) < \epsilon$, 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 $\mathcal{E}(\mathfrak{X})$ 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.

4.0.1 Python Implementation

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

```python
import numpy as np
from sklearn.utils import shuffle

def partition_data(x, y, train_ratio):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (m, N)
    y : array_like
        y.shape = (k, N)
    train_ratio : float
        0<=train_ratio<=1

    Returns
    -------
    train : Tuple[array_like]
    dev : Tuple[array_like]
    test : Tuple[array_like]
    """
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) #
    x = x.T
    y = y.T

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

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

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

    return train, dev, test
```

5 Regularization

Suppose we're training an L-layer neural network with dataset $\{(x_j, y_j)\} \subset \mathbb{R}^{s_0} \times \mathbb{R}^{s_L}$ with N examples. Assuming a generic loss function $\mathbb{L} : \mathbb{R}^{s_L} \times \mathbb{R}^{s_L} \to \mathbb{R}$, then we have our cost function $\mathbb{J}$ defined on our one-parameter families of parameters W and b given by

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

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

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

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

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

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

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

and

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

The idea behind regularization is that we're now minimizing

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

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

A typical usage of regularization would be similar to the following outline:

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

ii. Give a set Λ of potential regularization parameters.

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

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

which dependent on λ.

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

v. After finding $\mathcal{E}_\lambda(\mathfrak{X})$ 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.

5.0.1 Python Implementation

```python
import numpy as np

import utils
import activators

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

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

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

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

    Returns
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
    a = cache[’a’]
    w = params[’w’]
    L = len(a)
    aL = a[L - 1]

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

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

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

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

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

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

    ## Compute gradients
    dw = {}
    db = {}
    for l in range(1, L + 1):
        db[l] = (1 / n) * np.sum(delta[l], axis=1, keepdims=True)
        assert(db[l].shape == (w[l].shape[0], 1))
        dw[l] = (1 / n) * (delta[l] @ a[l - 1].T + lambda_ * w[l])
        assert(dw[l].shape == w[l].shape)
    grads ={'dw' : dw, 'db' : db}
    return grads


def model(x, y,
          hidden_layer_sizes,
          activators,
          lambda_=0.0,
          num_iters=1e4,
          print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    lambda_ : float
        The regularization parameter
        Default: 0.0
    num_iters : int
        Number of iterations with which our model performs gradient descent
        Default: 10000
    print_cost : Boolean
        If True, print the cost every 1000 iterations
        Default: False

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

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

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

    return params, cost
```

5.1 (Inverted) Dropout Regularization

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

$$\underbrace{\begin{bmatrix} x^1 \\ \vdots \\ x^{s_0} \end{bmatrix}}_{\text{Layer 0}} \xrightarrow{\varphi^{[1]}} \underbrace{\begin{bmatrix} z^{[1]1} \\ \vdots \\ z^{[1]s_1} \end{bmatrix} \xrightarrow{g^{[1]}} \begin{bmatrix} a^{[1]1} \\ \vdots \\ a^{[1]s_1} \end{bmatrix}}_{\text{Layer 1}} \xrightarrow{\varphi^{[2]}} \underbrace{\begin{bmatrix} z^{[2]1} \\ \vdots \\ z^{[2]s_2} \end{bmatrix} \xrightarrow{g^{[2]}} \begin{bmatrix} a^{[2]1} \\ \vdots \\ a^{[2]s_2} \end{bmatrix}}_{\text{Layer 2}} \xrightarrow{\varphi^{[3]}} \text{output},$$

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

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

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

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

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

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.

```python
import numpy as np

import utils

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

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



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

    Returns
    -------
    cache : Dict[Dict]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params[’w’]
    b = params[’b’]
    L = len(w) # Number of layers including input layer
    n = x.shape[1]

    # Set empty caches
    a = {}
    z = {}
    # Dropout on layer 0
    a[0] = x
    a[0] = a[0] * D[0]
    a[0] /= keep_prob[0]
    # Loop through hidden layers
    for l in range(1, L):
        zl, al = utils.linear_activation_forward(a[l - 1], w[l], b[l], activators[l
        al = al * D[l]
        al /= keep_prob[l]
        z[l] = zl
        a[l] = al

    # Output layer
    z[L], a[L] = utils.linear_activation_forward(a[L - 1], w[L], b[L], activators[-

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

def backward_propagation(x, y, params, cache, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer[l], num_ex)
        D[l] = a Boolean array
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

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

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

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

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

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

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

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

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

    return params, cost
```

5.2 Data Augmentation

This section requires work.

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

5.3 Early Stopping

This section requires work.

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

6 Gradients and Numerical Remarks

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

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

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

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

6.1 Numerical Gradient Checking

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

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

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

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

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

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

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

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

6.1.1 Python Implementation

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

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

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

7 Gradient Descent

So far in our implementation of gradient descent, we use the entire training set for every iteration of gradient descent. This method is called *batch gradient descent*. We modify this method, 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 $\mathfrak{X}$ with $|\mathfrak{X}| = n$, where n is very large (e.g., $n = 5000000$). We fix a batch size b (e.g., $b = 5000$), and partition $\mathfrak{X}$ into 1000 mini-batches

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

where $\left\lceil \frac{n}{b} \right\rceil$ denote the ceiling function. We then perform gradient descent in the following manner:

1. For $i \in [0, I)_{\mathbb{Z}}$ (where I denote the number of iterations to perform gradient descent):

   a. For $t \in \left[0, \left\lceil \frac{n}{b} \right\rceil\right)_{\mathbb{Z}}$:

      i. Perform forward propagation on $\mathfrak{X}^t$:

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

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

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

      iii. Perform backward propagation on $\mathfrak{X}^t$:

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

iv. Perform gradient descent:

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

We make several remarks about mini-batch gradient descent:

- Batch gradient descent doesn't always decrease (e.g., our learning rate is too large). Mini-batch may oscillate rapidly, but the general direction should move towards a minimum.

- If $b = n$, then we fully recover batch gradient descent. This is typically too computationally expensive since we use the full training set for each iteration.

- If $b = 1$, then we recover stochastic gradient descent, i.e., we train our model on a different example during each iteration. We lose all the speed related to vectorization, since we're dealing with single examples during each iteration.

- Choose $1 < b < n$ is typically always the best solution, since it deals with both of the aforementioned problems.

- Due to the nature of a computer's internal structure, it's typically better to choose a batch size b for the form

  $$b = 2^p,$$

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

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

7.0.1 Python Implementation

Appendices

A utils.py

```
#! python3
import copy

import numpy as np
from sklearn.utils import shuffle

import activators
from activators import ACTIVATORS

## Usefule printing function
def print_array_dict(D):
    """
    Parameters
    ----------
    D : Dict[array_like]
    Returns
    -------
    None
    """
    txt = "Array␣{0}␣has␣shape␣{1}\n{2}"
    for k, v in D.items():
        print(txt.format(str(k), v.shape, v))


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

    Returns
    -------
    train : Tuple[array_like]
    dev : Tuple[array_like]
    test : Tuple[array_like]
    """
    ## Shuffle the data
    x, y = shuffle(x.T, y.T) #
    x = x.T
    y = y.T

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

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

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

    return train, dev, test


##### General Neural Network Model #####

## Retrieve number of examples and layer dimensions
def dim_retrieval(x, y, hidden_sizes):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[L], n)
    hidden_sizes : List[int]
        hidden_sizes[i-1] = The number nodes layer i
    Returns
    -------
    n : int
        The number of training examples
    layers : List
        layer[l] = # nodes in layer l

    """
    m, n = x.shape
    assert(y.shape[1] == n)
    K = y.shape[0]
    layers = [m]
    layers.extend(hidden_sizes)
    layers.append(K)

    return n, layers

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

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

## Initialize parameters using the size of each layer
def initialize_parameters_random(layers):
    """
    Parameters
    ----------
    layers : List[int]
        layers[l] = # nodes in layer l
    Returns
    -------
    params : Dict[Dict]
        w[l] : array_like
            dwl.shape = (layers[l], layers[l-1])
        b[l] : array_like
            dbl.shape = (layers[l], 1)
    """
    w = {}
    b = {}
    for l in range(1, len(layers)):
        w[l] = np.random.randn(layers[l], layers[l - 1]) * 0.01
        b[l] = np.zeros((layers[l], 1))
    params = {'w' : w, 'b' : b}
    return params

## Forward and Backward Linear Activations
def linear_activation_forward(a_prev, w, b, activator):
    """
    Parameters
    ----------
    a_prev : array_like
        a_prev.shape = (layers[l], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    b : array_like
        b.shape = (layers[l+1], 1)
    activator : str
        activator = 'relu', 'sigmoid', or 'tanh'

    Returns
    -------
    z : array_like
        z.shape = (layer_dims[l+1], n)
    a : array_like
        a.shape = (layer_dims[l+1], n)
    """
    assert activator in ACTIVATORS, f'{activator}␣is␣not␣a␣valid␣activator.'

    z = w @ a_prev + b
    if activator == 'relu':
        a, _ = activators.relu(z)
    elif activator == 'sigmoid':
        a, _ = activators.sigmoid(z)
    elif activator == 'tanh':
        a, _ = activators.tanh(z)
    return z, a

def linear_activation_backward(delta_next, z, w, activator):
    """
    Parameters
    ----------
    delta_next : array_like
        delta_next.shape = (layers[l+1], n)
    z : array_like
        z.shape = (layers[l+1], n)
    w : array_like
        w.shape = (layers[l+1], layers[l])
    activator : str
        activator = ’relu’, ’sigmoid’, or ’tanh’

    Returns
    -------
    delta : array_like
        delta.shape = (layers[l], n)
    """
    assert activator in ACTIVATORS, f’{activator}␣is␣not␣a␣valid␣activator.’

    n = delta_next.shape[1]

    if activator == ’relu’:
        _, dg = activators.relu(z)
    elif activator == ’sigmoid’:
        _, dg = activators.sigmoid(z)
    elif activator == ’tanh’:
        _, dg = activators.tanh(z)

    da = w.T @ delta_next
    assert(da.shape == (w.shape[1], n))
    delta = da * dg
    assert(delta.shape == (w.shape[1], n))
    return delta

## Forward and Backward Propagation with Dropout Regularization
def forward_propagation(x, params, activators, D, keep_prob=1.0):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0] n)
    params : Dict[Dict]
        params[’w’][l] : array_like
            wl.shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            bl.shape = (layers[l], 1)
    activators : List[str]
        activators[l] = activation function of layer l+1
    D : Dict[array_like]
        D[l].shape = (layer_dims[l], num_ex)
        D[l] = a Boolean array astype(int)
    keep_prob : List[float]
        keep_prob[l] = The probabilty of keeping a node in layer l

    Returns
    -------
    cache : Dict[Dict]
        cache[’z’][l] : array_like
            z[l].shape = (layers[l], n)
        cache[’a’][l] : array_like
            a[l].shape = (layers[l], n)
    """
    # Retrieve parameters
    w = params[’w’]
    b = params[’b’]
    L = len(w) # Number of layers excluding output layer
    n = x.shape[1]
    # Set empty caches
    a = {}
    z = {}
    # Dropout on layer 0
    a[0] = x
    a[0] = a[0] * D[0]
    a[0] /= keep_prob[0]
    # Loop through hidden layers
    for l in range(1, L + 1):
        zl, al = linear_activation_forward(a[l - 1], w[l], b[l], activators[l - 1])
        al = al * D[l]
        al /= keep_prob[l]
        z[l] = zl
        a[l] = al
    # Output layer
    z[L], a[L] = linear_activation_forward(a[L - 1], w[L], b[L], activators[-1])

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

def backward_propagation(x, y, params, cache, activators, D, keep_prob):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    params : Dict[Dict[array_like]]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    cache : Dict[Dict[array_like]]
```

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

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

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

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

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

```

## Compute the cost
def compute_cost(y, cache):
    """
    Parameters
    ----------
    y : array_like
        y.shape = (layers[-1], n)
    cache : Dict[Dict]
        cache['z'][l] : array_like
            z[l].shape = (layers[l], n)
        cache['a'][l] : array_like
            a[l].shape = (layers[l], n)
    -------
    cost : float
        The cost evaluated at y and aL
    """
    ## Retrieve parameters
    n = y.shape[1]
    a = cache['a']
    L = len(a)
    aL = a[L - 1]

    cost = (-1 / n) * (np.sum(y * np.log(aL)) + np.sum((1 - y) * np.log(1 - aL)))
    cost = float(np.squeeze(cost))

    return cost

## Update parameters via gradient descent
def update_parameters(params, grads, learning_rate=0.01):
    """
    Parameters
    ----------
    params : Dict[Dict]
        params['w'][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params['b'][l] : array_like
            b[l].shape = (layers[l], 1)
    grads : Dict[Dict]
        grads['dw'][l] : array_like
            dw[l].shape = w[l].shape
        grads['db'][l] : array_like
            db[l].shape = b[l].shape
    learning_rate : float
        Default: 0.01
        The learning rate for gradient descent

    Returns
    -------
    params : Dict[Dict]
        params[’w’][l] : array_like
            w[l].shape = (layers[l], layers[l-1])
        params[’b’][l] : array_like
            b[l].shape = (layers[l], 1)
    """
    ## Retrieve parameters
    w = copy.deepcopy(params[’w’])
    b = copy.deepcopy(params[’b’])
    L = len(w)

    ## Retrieve gradients
    dw = grads[’dw’]
    db = grads[’db’]

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

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

def model_nn(x, y,
          hidden_layer_sizes,
          activators,
          keep_prob=1.0,
          num_iters=10000,
          print_cost=False):
    """
    Parameters
    ----------
    x : array_like
        x.shape = (layers[0], n)
    y : array_like
        y.shape = (layers[-1], n)
    hidden_layer_sizes : List[int]
        The number nodes layer l = hidden_layer_sizes[l-1]
    activators : List[str]
        activators[l] = activation function of layer l+1
    keep_prob : List[float] | float
        keep_prob[l] = The probabilty of keeping a node in layer l
        keep_prob = The same probability for all input and hidden layers
    num_iters : int
        Number of iterations with which our model performs gradient descent
    print_cost : Boolean
        If True, print the cost every 1000 iterations

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

    ## Expand keep_prob to a list if it's a single float
    if isinstance(keep_prob, float):
        keep_prob = [keep_prob] * (len(layers) - 1)

    # main gradient descent loop
    for i in range(num_iters):
        D = dropout_matrices(layers, n, keep_prob)
        cache = forward_propagation(x, params, activators, D, keep_prob)
        cost = compute_cost(cache, y)
        grads = backward_propagation(x, y, params, cache, activators, D, keep_prob)
        params = update_parameters(params, grads)

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

    return params, cost







########## TESTING ##########
def test_dropout_nn():
    x = np.random.rand(4, 500)
    y = np.random.rand(1, 500)
    hidden_layer_sizes = [4, 5, 4]
    activators = ['relu', 'relu', 'relu', 'sigmoid']
    keep_prob = 1.0
    params, cost = model_nn(x, y, hidden_layer_sizes, activators, keep_prob)
    print(params)



######## Functions to use later
def reshape_labels(num_labels, y):
    """
    Parameters
    ----------
    num_labels : int
        The number of possible labels the output y may take
    y : array_like
        y.size = n
        y[i] takes values in {1,2,...,num_labels}
    Returns
    Y : array_like
        Y.shape = (num_lables, n)
        Y[i][j] = 1 if y[j] = i, Y[i][j] = 0 otherwise
    -------
    """

    if num_labels <= 2:
        return y
    else:
        omega = []
        for i in range(num_labels):
            omega.append(np.eye(1, num_labels, i))  # the standard i-th basis vecto

        Y = np.concatenate([omega[i] for i in y], axis=0).T
        for i in range(num_labels):
            for j in range(n):
                if y[j] == i:
                    assert Y[i][j] == 1
                else:
                    assert Y[i][j] == 0
        return Y

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

B activators.py

```
import numpy as np

```

```
ACTIVATORS = [’relu’, ’sigmoid’, ’tanh’]

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

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

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

    Returns
    -------
    sigma : array_like
        The (broadcasted) value of the sigmoid function evaluated at z
    dsigma : array_like
        The (broadcasted) derivative of the sigmoid function evaluate at z
    """
    # Compute value of sigmoid
    sigma = (1 / (1 + np.exp(-z)))
    # Compute differential of sigmoid
    dsigma = sigma * (1 - sigma)
    return sigma, dsigma

# The hyperbolic tangent function
```

```python
def tanh(z):
    """
    Parameters
    ----------
    z : array_like

    Returns
    phi : array_like
        The (broadcasted) value of the hyperbolic tangent function evaluated at z
    dphi : array_like
        The (broadcasted) derivative of hyperbolic tangent function evaluated at z
    """
    # Compute value of tanh
    phi = np.tanh(z)
    # Compute differential of tanh
    dphi = 1 - (phi * phi)
    return phi, dphi
```