Neural Networks

Matt R

February 22, 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
```

```python
            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 now we wish to consider the binary classification problem given the training set (x, y) with $x \in \mathbb{R}^{m\times n}$ and $y \in \{0,1\}^n$. Usually with logistic regression we have the following network:

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

where

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

is our affine-linear transformation, and

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

is our sigmoid function. Logistic regression can be too simplistic of a model for many situations. To modify this model to handle more complex situations, we introduce a new "hidden layer" of nodes with activation functions. That is, we consider a network of the following form:

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

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 wehere 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).

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:

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

$$g^{[1]} : \mathbb{R}^{s_1} \to \mathbb{R}^{s_1}, \qquad dg^{[1]} : T\mathbb{R}^{s_1} \to T\mathbb{R}^{s_1},$$

$$a^{[1]} = g^{[1]}(z^{[1]}), \qquad dg^{[1]}_{z^{[1]}}(v) = \begin{bmatrix} g^{[1]\prime}(z^{[1]1}) & 0 & \cdots & 0 \\ 0 & g^{[1]\prime}(z^{[1]2}) & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & g^{[1]\prime}(z^{[1]s_1}) \end{bmatrix} \begin{bmatrix} v^1 \\ v^2 \\ \vdots \\ v^{s_1} \end{bmatrix};$$

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

$$\begin{aligned} g^{[2]} &: \mathbb{R} \to \mathbb{R}, & dg^{[2]} &: T\mathbb{R} \to T\mathbb{R}, \\ a^{[2]} &= g^{[2]}(z^{[2]}), & dg^{[2]}_{z^{[2]}}(v) &= g^{[2]\prime}(z^{[2]}) \cdot v. \end{aligned}$$

That is, given an input $x \in \mathbb{R}^{s_1}$, we get a predicted value $\hat{y}$ 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.*

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

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

We now define the compositional function

$$F : \mathbb{R}^{s_0} \to \mathbb{R}, \qquad F(x) = \mathbb{L}_y \circ g^{[2]} \circ \varphi^{[2]} \circ g^{[1]} \circ \varphi^{[1]}(x).$$

As we mentioned before, we wish to optimize with respect to our parameters, but our above composition doesn't make this dependence explicit for computations. To this end, we consider the generic affine-linear transformation

$$\varphi : \mathbb{R}^m \to \mathbb{R}^k, \qquad \varphi(x) = Wx + b,$$

with $W \in \mathbb{R}^{k\times m}, b \in \mathbb{R}^k$, and instead consider the related

$$\phi : \mathbb{R}^{k\times m} \times \mathbb{R}^k \to \mathbb{R}^k, \qquad \phi(W, b) = Wx + b,$$

for some fixed $x \in \mathbb{R}^m$. Then we see that

$$d\phi : T\mathbb{R}^{k\times m} \times T\mathbb{R}^k \to T\mathbb{R}^k,$$

$$\begin{aligned}
d\phi_{(W,b)}(V,v) &= \frac{d}{dt}\bigg|_{t=0} \phi(W+tV, b+tv) \\
&= \frac{d}{dt}\bigg|_{t=0} (W+tV)x + (b+tv) \\
&= Vx + v \\
&= \begin{bmatrix} x & 1 \end{bmatrix} \begin{bmatrix} V \\ v \end{bmatrix}.
\end{aligned}$$

Taking this further, we now consider the map

$$\Phi : \mathbb{R}^{k\times m} \times \mathbb{R}^k \times \mathbb{R}^m \to \mathbb{R}^k, \qquad \Phi(W,b,x) = Wx + b,$$

and then computing our differential for

$$d\Phi : T\mathbb{R}^{k\times m} \times T\mathbb{R}^k \times T\mathbb{R}^m \to T\mathbb{R}^k,$$

yields

$$\begin{aligned}
d\Phi_{(W,b,x)}(V,v,p) &= \frac{d}{dt}\bigg|_{t=0} \Phi(W+tV, b+tv, x+tp) \\
&= \frac{d}{dt}\bigg|_{t=0} ((W+tV)(x+tp) + (b+tv)) \\
&= \frac{d}{dt}\bigg|_{t=0} \big(Wx + tVx + tWv + t^2Vp + b + tv\big) \\
&= Vx + Wp + v \\
&= \begin{bmatrix} x & 1 & W \end{bmatrix} \begin{bmatrix} V \\ v \\ p \end{bmatrix} \\
&= d\phi_{(W,b)}(V,v) + d\varphi_x(p)
\end{aligned}$$

This function Φ is what we want in our compositional-function, and so we redefine F as

$$F(W^{[2]}, b^{[2]}, W^{[1]}, b^{[1]}, x) = \mathbb{L}_y \circ g^{[2]} \circ \Phi^{[2]} \circ (W^{[2]}, b^{[2]}, g^{[1]} \circ \Phi^{[1]} \circ (W^{[1]}, b^{[1]}, x))$$

Taking the exterior derivative, and noting the composition turns into matrix multiplication on the tangent space, we get

$$\begin{aligned}
&dF_{(W^{[2]},b^{[2]},W^{[1]},b^{[1]},x)}(U,u,V,v,p) \\
&\quad = d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot d\Phi^{[2]}_{(W^{[2]},b^{[2]},a^{[1]})} \cdot (U,u,dg^{[1]}_{z^{[1]}} \cdot d\Phi^{[1]}_{(W^{[1]},b^{[1]},x)}(V,v,p)) \\
&\quad = d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot (d\phi^{[2]}_{(W^{[2]},b^{[2]})}(U,u) + d\varphi^{[2]}_{a^{[1]}} \cdot dg^{[1]}_{z^{[1]}} \cdot (d\phi^{[1]}_{(W^{[1]},b^{[1]})}(V,v) + d\varphi^{[1]}_x(p))) \\
&\quad = d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot d\phi^{[2]}_{(W^{[2]},b^{[2]},\{a^{[1]}\})}(U,u) \\
&\qquad + d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot d\varphi^{[2]}_{a^{[1]},\{W^{[2]},b^{[2]}\}} \cdot dg^{[1]}_{z^{[1]}} \cdot d\phi^{[1]}_{(W^{[1]},b^{[1]}),\{x\}}(V,v) \\
&\qquad + d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot d\varphi^{[2]}_{a^{[1]},\{W^{[2]},b^{[2]}\}} \cdot dg^{[1]}_{z^{[1]}} \cdot d\varphi^{[1]}_{x,\{W^{[1]},b^{[1]}\}}(p) \qquad =: dF^{[2]} + dF^{[1]} +
\end{aligned}$$

Recalling that the gradient is the transpose of the exterior derivative in Euclidean space, we then conclude that

$$\begin{aligned}
\nabla F(x) &= (dF_x)^T \\
&= (d(\mathbb{L}_y)_{a^{[2]}} \cdot dg^{[2]}_{z^{[2]}} \cdot d\varphi^{[2]}_{a^{[1]}} \cdot dg^{[1]}_{z^{[1]}} \cdot d\varphi^{[1]}_x)^T \\
&= (d\varphi^{[1]}_x)^T \cdot (dg^{[1]}_{z^{[1]}})^T \cdot (d\varphi^{[2]}_{a^{[1]}})^T \cdot (dg^{[2]}_{z^{[2]}})^T \cdot (d(\mathbb{L}_y)_{a^{[2]}})^T \\
&= \nabla\varphi^{[1]}(x) \cdot \nabla g^{[1]}(z^{[1]}) \cdot \nabla\varphi^{[2]}(a^{[1]}) \cdot \nabla g^{[2]}(z^{[2]}) \cdot \nabla\mathbb{L}_y(a^{[2]})
\end{aligned}$$

This auxiliary computation is nice, but isn't quite what we're looking for since we would like to compute the differentials with respect to our weight parameters $W^{[\ell]}$ and $b^{[\ell]}$.