Deep Learning in Python

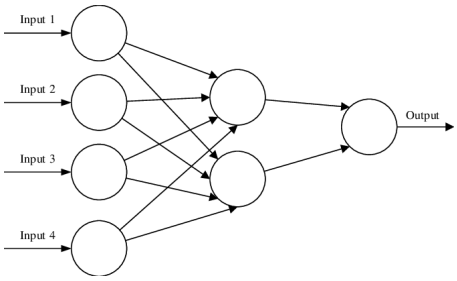
Contents

1 Neural Networks and Deep Learning

1.1 Logistic Regression as a Neural Network

1.1.1 Introduction to Logistic Regression

A single **neural network** can be built off of an input (x), an activation layer known as a *neuron*, and producing an output (y). A larger neural network is then formed by taking many of the single neurons and stacking them together. Each *feature* $(x_1, x_2, ..., x_n)$ can be used as an input to the activation layers to produce our output y. For the below example, we say that the layer in the middle is *densely connected* since every feature is in input.



To store an image, your computer stores three different matrices corresponding to the red, green, and blue channel (RGB values). So if your input image is 64x64 pixels, you will have three 64x64 matrices. To unroll these values into a **feature vector**, we will add values from all 3 vectors into a single x vector, where n_x is the number of features in the vector (in this case, 12288).

In **binary classification**, our goal is to learn a classifier that can input an image represented by feature vector x and predict whether the corresponding label y is a 1 or 0 (1 for cat, 0 for non cat).

Here is some common notation we will be using:

- Single training example: (x,y) where $x \in \mathbb{R}^{n_x}$, y $\in \{0, 1\}$
- M training examples: $\{(x^1, y^1), ..., (x^m, y^m)\}$
- Matrix X: $\begin{bmatrix} | & | & ... & | \\ x^1 & x^2 & ... & x^m \\ | & | & ... & | \end{bmatrix}$ where rows = n_x, columns = m. In Python, X.shape = (n_x, m).
- Matrix Y: $\begin{bmatrix} y^1, y^2, ..., y^m \end{bmatrix}$ where Y.shape = (1, m)

Given x, we want an estimate known as $\hat{y}$ = P(y=1|x) given the following parameters: $x \in \mathbb{R}^{n_x}$, $w \in \mathbb{R}^{n_x}$, and $b \in \mathbb{R}$. We want out output to be $0 \leq \hat{y} \leq 1$, so we will use the **sigmoid function** to find our output, which will be:

$$\hat{y} = \sigma(z) \text{ where } z = w^T x + b \text{ and } \sigma(z) = \frac{1}{1 + e^{-z}}$$

If z is large, then $\sigma(z)$ will be very close to 1. But if z is a large negative number, then $\sigma(z)$ will be very close to 0. So given $\{(x^1, y^1), ..., (x^m, y^m)\}$ we want $\hat{y}^i \approx y^i$

1.1.2 Logistic Regression Cost Function

We will associate x^i, y^i, and z^i with the i^{th} training example of our data. We will need to define a **loss function**, with respect to a single training example, to measure how good our output ($\hat{y}$) is when the true label is y. Since we are using gradient descent, we will define the following loss function:

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

We want this loss function to be as small as possible. Lets look at the two cases:

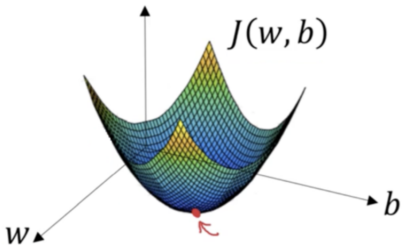
If y=1: $\mathcal{L}(\hat{y}, y) = -y\, log(\hat{y})$ and we want this to be as small as possible ($\hat{y}$ large).
If y=0: $\mathcal{L}(\hat{y}, y) = -\, log(1 - \hat{y})$ and we want this to be large ($\hat{y}$ small).

To train the parameters w and b, we need to define a **cost function**, which measures how well your doing on an entire training set (cost of the parameters). We will define this as:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^i, y^i) = -\frac{1}{m}\sum_{i=1}^{m}[(y^i\, log(\hat{y}^i) + (1-y^i)\, log(1-\hat{y}^i))]$$

1.1.3 Gradient Descent

We want to find the values of w and b that will *minimize* the cost function J(w,b). Our cost function that we defined is convex (only one minimum). So we will initialize (w,b) to a random value, typically zero, and will take steps downhill in the steepest direction it can. Eventually it will converge to a minimum value and find our parameter values.



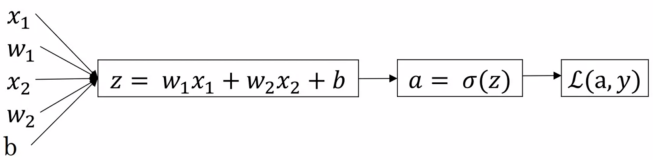
Gradient descent will repeatedly update the value of w and b with the formula:

$$w = w - \alpha\frac{\partial J(w,b)}{\partial w}\ ,\ b = b - \alpha\frac{\partial J(w,b)}{\partial b}$$

where α is our learning rate that we set multiplied by the partial derivative (since there are two variables) of the cost function with respect to the given parameter.

We want to modify out w and b parameters in order to reduce the loss when performing gradient descent on our Logistic Regression. We can set up a computation graph to find the derivatives through **backpropagation**. We will do this for a *single* training example, lets remind ourselves of our equations and the graph:

$z = w^T x + b$
$\hat{y} = a = \sigma(z)$
$\mathcal{L}(a,y) = -(y\, log(a) + (1-y)\, log(1-a))$



The first back step is to compute "da" $= \frac{\partial\mathcal{L}(a,y)}{\partial a} = \frac{-y}{a} - \frac{1-y}{1-a}$

Next we step back again and compute "dz" $= \frac{\partial\mathcal{L}(a,y)}{\partial z} = \frac{\partial\mathcal{L}}{\partial a} * \frac{\partial a}{\partial z} = a(1-a) * (\frac{-y}{a} + \frac{1-y}{1-a}) = a - y$

The final step back is to find how much to change our w_1, w_2, and b values. We can do this by:

- Calculating: $\frac{\partial\mathcal{L}}{\partial w_1}$ = "dw_1" $= x_1 * dz$, "dw_2" $= x_2 * dz$, and "db" $= dz$
- Then update our variables: $w_1 = w_1 - \alpha * dw_1$, $w_2 = w_2 - \alpha * dw_2$, and $b = b - \alpha * dz$

Now we want to **perform gradient descent on m examples**. This will use the cost function (not the loss function like we did on a single example). We will write sudo-code for Python that implements this m example gradient descent:

Initialize: J=0, dw_1=0, dw_2=0, db=0
for i=1 to m:
$z^i = w^T x^i + b$
$z^i = \sigma(z^i)$
$J+ = -[(y^i\, log(a^i) + (1-y^i)\, log(1-a^i))]$
$dz^i = a^i - y^i$
$dw_1+ = x_1^i * dz^i$
$dw_2+ = x_2^i * dz^i$
$db+ = dz^i$

After the loop, we then take the average and update our varaibles:

J /= m
dw_1 /= m
dw_2 /= m
db /= m
$w_1 = w_1 - \alpha * dw_1$
$w_2 = w_2 - \alpha * dw_2$
$b = b - \alpha * db$

1.2 Vectorization in Python