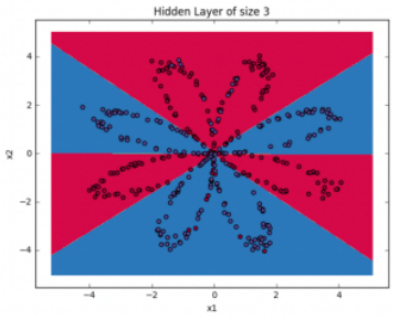
Week 3 - prog ex: Back-propagation (Planar data classification with a hidden layer)

笔记本: DL 1 - NN and DL
创建时间: 2021/1/8 11:59 **更新时间:** 2021/1/8 12:52

Welcome to the second programming exercise of the deep learning specialization. In this notebook you will generate red and blue points to form a flower. You will then fit a neural network to correctly classify the points. You will try different layers and see the results.



By completing this assignment you will:

- Develop an intuition of back-propagation and see it work on data.

- Recognize that the more hidden layers you have the more complex structure you could capture.

- Build all the helper functions to implement a full model with one hidden layer.

Planar data classification with one hidden layer

Welcome to your week 3 programming assignment. It's time to build your first neural network, which will have a hidden layer. You will see a big difference between this model and the one you implemented using logistic regression.

You will learn how to:

- Implement a 2-class classification neural network with a single hidden layer
- Use units with a non-linear activation function, such as tanh
- Compute the cross entropy loss
- Implement forward and backward propagation

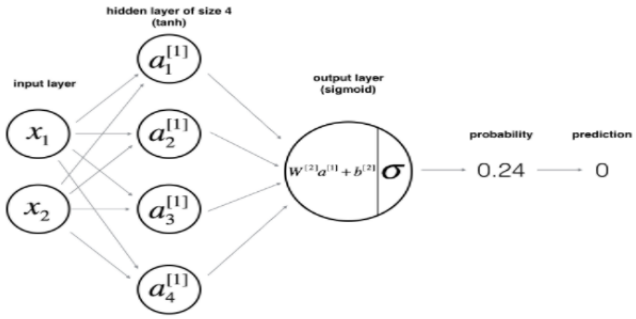
sklearn Logistic Regression:

```
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);

# Plot the decision boundary for logistic regression
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")


# Print accuracy
LR_predictions = clf.predict(X.T)
```

```
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) +
np.dot(1-Y,1-LR_predictions))/float(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

NN model

Here is our model:



Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]} \tag{1}$$
$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]} \tag{3}$$
$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$
$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right) \tag{6}$$

Cost

Now that you have computed $A^{[2]}$ (in the Python variable "A2"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

$$J = -\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log\left(a^{[2](i)}\right) + (1 - y^{(i)}) \log\left(1 - a^{[2](i)}\right) \right) \tag{13}$$

Exercise: Implement `compute_cost()` to compute the value of the cost J.

Instructions:

- There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)} \log(a^{[2](i)})$:

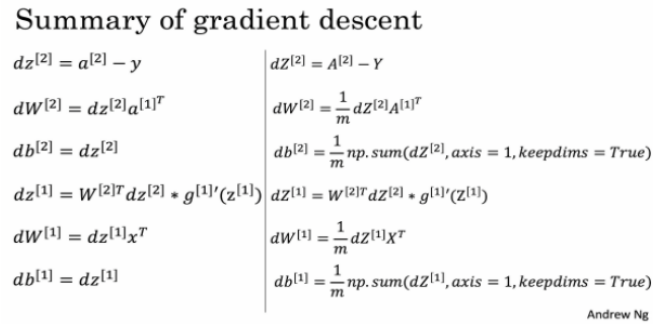
```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

```python
def compute_cost(A2, Y, parameters):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 — The sigmoid output of the second activation, of shape (1, number of examples)
    Y — "true" labels vector of shape (1, number of examples)
    parameters — python dictionary containing your parameters W1, b1, W2 and b2
    [Note that the parameters argument is not used in this function,
    but the auto-grader currently expects this parameter.
    Future version of this notebook will fix both the notebook
    and the auto-grader so that `parameters` is not needed.
    For now, please include `parameters` in the function signature,
    and also when invoking this function.]

    Returns:
    cost — cross-entropy cost given equation (13)

    """

    m = Y.shape[1] # number of example

    # Compute the cross-entropy cost
    ### START CODE HERE ### (≈ 2 lines of code)
    logprobs = np.multiply(Y, np.log(A2)) + np.multiply(1-Y, np.log(1-A2))
    cost = -np.sum(logprobs,axis=1)/m
    ### END CODE HERE ###

    cost = float(np.squeeze(cost))  # makes sure cost is the dimension we expect.
                                    # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost
```

GD

Question: Implement the function `backward_propagation()`.

Instructions: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.



- Tips:
  - To compute dZ1 you'll need to compute $g^{[1]\prime}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]\prime}(z) = 1 - a^2$. So you can compute $g^{[1]\prime}(Z^{[1]})$ using `(1 - np.power(A1, 2))`.

```
    # Backward propagation: calculate dW1, db1, dW2, db2.
    ### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)
    dZ2 = A2-Y
    dW2 = np.dot(dZ2,A1.T)/m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)/m
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1-np.power(A1, 2))
    dW1 = np.dot(dZ1, X.T)/m
    db1 = np.sum(dZ1, axis = 1, keepdims = True)/m
    ### END CODE HERE ###

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

Interpretation:

- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
- The best hidden layer size seems to be around n_h = 5. Indeed, a value around here seems to fits the data well without also incurring noticeable overfitting.
- You will also learn later about regularization, which lets you use very large models (such as n_h = 50) without much overfitting.

Optional questions:

Note: Remember to submit the assignment by clicking the blue "Submit Assignment" button at the upper-right.

Some optional/ungraded questions that you can explore if you wish:

- What happens when you change the tanh activation for a sigmoid activation or a ReLU activation?
- Play with the learning_rate. What happens?
- What if we change the dataset? (See part 5 below!)