Regularization

In the last lesson, we saw the importance of the weak Deep Learning models tends to much flexibility and capacity that **overfitting can be a serious problem**. Fortunately, regularization is a big thing such. Even in the beginning, we can do many tricks to avoid **overfitting and generalize to new examples** that Deep Learning has been done.

Dropout is one regularization in your deep learning models.

Let's learn how the packages we are going to use.

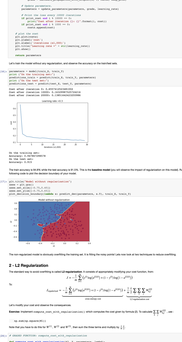
```
In [1]: # Import packages
        import numpy as np
        import matplotlib.pyplot as plt
        from keras.models import Sequential, Graph
        from keras.layers.core import Dense, Dropout, Activation
        from keras.layers.embeddings import Embedding
        from keras.layers.recurrent import SimpleRNN, LSTM
        Using Theano backend.
        WARNING: theano.sandbox.cuda failed to import
        Using gpu device 0: GeForce GT 640 (Kepler) (CNMeM is disabled, cuDNN not available)
        /home/ubuntu/anaconda2/lib/python2.7/site-packages/theano/sandbox/cuda/__init__.p
        y:397: UserWarning: You are trying to use the{ cuDNN backend but it is not detected
        in the{ ldconfig path (or the{ CUDNN_{ROOT_{DIR environment variable). This{ will
        make{ compilation{ slower.
        warnings.warn("You are trying{ to use the{ cuDNN backend but it is not detected{
        in the{ ldconfig path (or the{ CUDNN_{ROOT_{DIR environment variable). This{ will
        make{ compilation{ slower.")
```

Problem Statement: You have a set of pictures from the Facebook Timeline. You would like us to recommend positions where Francis's goal keeper should kick the ball so that the Front team is prepare get their ball in their head.

Figure 1 : Football field

The goal keeper kicks the ball in the air, the players of each team are fighting to hit the ball with their head

```
In [2]: # Load the dataset but only keep the top n words, zero the rest
        # X is shape (m, 2) and Y is shape (m, 1)
        from reg_utils import load_2D_dataset
        train_X, train_Y, test_X, test_Y = load_2D_dataset()
```



Each dot corresponds to a position on the football field where a football player has hit the ball with his head after the French goal keeper has shot the ball from the left side of the football field.

- In the dataset, is there the French player targeted to hit their head with his/her head
- Our goal is to find the French team's goal keeper should kick the ball.

The goal: Use a deep learning model to find a position on the field where the goalkeeper should kick the ball.

Analysis of the dataset: This dataset is a little noisy, but it looks like a diagonal line separating the upper left half (blue) from the lower right half (red) would work well.

You will find by a non-regularized model. Then you'll learn how to regularize it and decide which model you will choose to solve the French Football Department's problem.

1 - Non-regularized model

You will use the following neural network (already implemented for you below). This model can be used:

- in regularization mode -- by setting the `lambd` input to a non-zero value. We use "`lambd`" instead of "`lambda`" because "`lambda`" is a reserved keyword in Python.
- in dropout mode -- by setting the `keep_prob` to a value less than one

You will first try the model without any regularization. Then, you'll implement:

- `compute_cost_with_regularization()` which computes a cost function that includes L2 regularization
- `backward_propagation_with_regularization()` which extends the gradients with respect to cost with L2 regularization.

A quick reminder: recall that the cost function with L2 regularization is:

$$J = -\frac{1}{m} \sum\limits_i \left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1- a^{[L](i)}\right) \right) \tag{1}$$

$$\text{cost with regularization} = \frac{\lambda}{2m} \sum\limits_l\sum\limits_k\sum\limits_j W_{k,j}^{[l]2} \tag{2}$$

Let's modify your cost and observe the consequences.

Exercise: Implement `compute_cost_with_regularization()` which computes the cost given by formula (2). To calculate $\sum\limits_k\sum\limits_j W_{k,j}^{[l]2}$, use:

```
np.sum(np.square(W))
```

Note that you have to divide the "W" by "m" and "2". Also use the three terms and multiply by $\frac{\lambda}{2m}$ ||

```
# GRADED FUNCTION: compute_cost_with_regularization

def compute_cost_with_regularization(A3, Y, parameters, lambd):
    """
    Implement the cost function with L2 regularization. See formula (2) above.

    Arguments:
    A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
    Y -- "true" labels vector, same shape as A3
    parameters -- python dictionary containing parameters of the model

    Returns:
    cost - value of the regularized loss function (formula (2))
    """
    m = Y.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]

    cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

    ### START CODE HERE ### (do not modify anything in the{ cross_entropy{ cost)
    L2_regularization_cost = np.sum(np.square(W1))/(m*4)+np.sum(np.square(W2))/(m*4)+np.sum(np.s
    quare(W3))/(m*4)

    cost = cross_entropy_cost + L2_regularization_cost
    ### END CODE HERE ###

    return cost
```

```
In [3]: A3, Y_assess, parameters = compute_cost_with_regularization_test_case()
        print("cost = " + str(compute_cost_with_regularization(A3, Y_assess, parameters, lambd = 0.1)))
        cost = 1.786648001
```

Expected Output:

cost = 1.786648001

(It's ok if you get a slightly different value here.)

Of course, because you changed the cost, you have to change backward propagation as well. All the gradients have to be computed with respect to this new cost.

Exercise: Implement the changes needed to backward propagation to take into account regularization. The changes only concern dW1, dW2 and dW3. For each, you have to add the regularization gradient ($\frac{\lambda}{m} W$) to the corresponding gradient.

```
In [4]: # GRADED FUNCTION: backward_propagation_with_regularization

        def backward_propagation_with_regularization(X, Y, cache, lambd):
            """
            Implements the backward propagation of our neural network.

            Arguments:
            X -- input dataset, of shape (input size, number of examples)
            Y -- "true" labels vector, of shape (output size, number of examples)
            cache -- cache output from forward_propagation()
            lambda -- regularization hyperparameter, scalar
            """
            m = X.shape[1]
            (z1, a1, W1, b1, z2, a2, W2, b2, z3, a3, W3, b3) = cache

            dz3 = a3 - Y
            dW3 = 1./m * np.dot(dz3, a2.T) + (lambd/(m*2))*W3
            db3 = 1./m * np.sum(dz3, axis=1, keepdims = True)

            dz2 = np.dot(W3.T, dz3) * (1 - a2**2)
            dW2 = 1./m * np.dot(dz2, a1.T) + (lambd/(m*2))*W2
            db2 = 1./m * np.sum(dz2, axis=1, keepdims = True)

            dz1 = np.dot(W2.T, dz2) * (1 - a1**2)
            dW1 = 1./m * np.dot(dz1, X.T) + (lambd/(m*2))*W1
            db1 = 1./m * np.sum(dz1, axis=1, keepdims = True)

            gradients = {"dz3": dz3, "dW3": dW3, "db3": db3,"da2": a2,
                         "dz2": dz2, "dW2": dW2, "db2": db2,"da1": a1,
                         "dz1": dz1, "dW1": dW1, "db1": db1}
            return gradients
```

```
In [5]: X_assess, Y_assess, cache = backward_propagation_with_regularization_test_case()
        grads = backward_propagation_with_regularization(X_assess, Y_assess, cache, lambd = 0.1)
        print ("dW1 = "+ str(grads["dW1"]))
        print ("dW2 = "+ str(grads["dW2"]))
        print ("dW3 = "+ str(grads["dW3"]))
        dW1 = [[-0.25604646 0.12419418 0.23789701]
        [ 0.31686832 -0.26022835 -0.44829085]]
        dW2 = [[-0.79592188 -0.16316055 0.73723527]
        [-0.81532281 -0.10053272 0.10099491]]
        dW3 = [[-0.27403627 0.06314963 0.26788808]]
```

Expected Output:

dW1 = [[-0.25604646 0.12419418 0.23789701]
[0.31686832 -0.26022835 -0.44829085]]
dW2 = [[-0.79592188 -0.16316055 0.73723527]
[-0.81532281 -0.10053272 0.10099491]]
dW3 = [[-0.27403627 0.06314963 0.26788808]]

Let's run the model with L2 regularization ($\lambda = 0.7$). The model is trained with dev set.

```
In [6]: # Train the model.
        parameters = model(train_X, train_Y, lambda = 0.7, initialization = "he", is_plot=Tr
        ue)
        print("On training set:")
        predictions_train = predict(train_X, train_Y, parameters)
        print("On test set:")
        predictions_test = predict(test_X, test_Y, parameters)
        Cost after iteration 0: 0.6974484493131264
        Cost after iteration 10000: 0.2684918873282239
        Cost after iteration 20000: 0.2268455926201365
```

```
        On this regularized model, the decision boundary is much better than the previous or without regularization.
        On training set:
        Accuracy: 0.938388625592
        On test set:
        Accuracy: 0.93
```

The model is now much smoother and less sensitive to noisy points.

2 - L2 Regularization

The standard way to avoid overfitting is called **L2 regularization**. It consists of appropriately modifying your cost function, from:

$$J = -\frac{1}{m} \sum\limits_i \left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1- a^{[L](i)}\right) \right) \tag{1}$$

$$\text{cost with regularization} = \frac{\lambda}{2m} \sum\limits_l\sum\limits_k\sum\limits_j W_{k,j}^{[l]2} \tag{2}$$

Let's modify your cost and observe the consequences.

Exercise: Implement `compute_cost_with_regularization()` which computes the cost given by formula (2). To calculate $\sum\limits_k\sum\limits_j W_{k,j}^{[l]2}$, use:

```
In [7]: # GRADED FUNCTION: compute_cost_with_regularization

        def compute_cost_with_regularization(A3, Y, parameters, lambd):
            """
            Implement the cost function with L2 regularization. See formula (2) above.

            Arguments:
            A3 -- post-activation, output of forward propagation, of shape (output size, number of examples)
            Y -- "true" labels vector, same shape as A3
            parameters -- python dictionary containing parameters of the model

            Returns:
            cost - value of the regularized loss function (formula (2))
            """
            m = Y.shape[1]
            W1 = parameters["W1"]
            W2 = parameters["W2"]
            W3 = parameters["W3"]

            cross_entropy_cost = compute_cost(A3, Y) # This gives you the cross-entropy part of the cost

            ### START CODE HERE ### (do not modify anything in the{ cross_entropy{ cost)
            L2_regularization_cost = np.sum(np.square(W1))/(m*4)+np.sum(np.square(W2))/(m*4)+np.s
            um(np.square(W3))/(m*4)

            cost = cross_entropy_cost + L2_regularization_cost
            ### END CODE HERE ###

            return cost
```

```
In [8]: A3, Y_assess, parameters = compute_cost_with_regularization_test_case()
        print("cost = " + str(compute_cost_with_regularization(A3, Y_assess, parameters, lambd = 0.1)))
        cost = 1.786648001
```

Expected Output:

cost = 1.786648001

(It's ok if you get a slightly different value here.)

Of course, because you changed the cost, you have to change backward propagation as well. All the gradients have to be computed with respect to this new cost.

Exercise: Implement the changes needed to backward propagation to take into account regularization. The changes only concern dW1, dW2 and dW3. For each, you have to add the regularization gradient ($\frac{\lambda}{m} W$) to the corresponding gradient.

```
In [9]: # GRADED FUNCTION: backward_propagation_with_regularization

        def backward_propagation_with_regularization(X, Y, cache, lambd):
            """
            Implements the backward propagation of our neural network.

            Arguments:
            X -- input dataset, of shape (input size, number of examples)
            Y -- "true" labels vector, of shape (output size, number of examples)
            cache -- cache output from forward_propagation()
            lambda -- regularization hyperparameter, scalar
            """
            m = X.shape[1]
            (z1, a1, W1, b1, z2, a2, W2, b2, z3, a3, W3, b3) = cache

            dz3 = a3 - Y
            dW3 = 1./m * np.dot(dz3, a2.T) + (lambd/(m*2))*W3
            db3 = 1./m * np.sum(dz3, axis=1, keepdims = True)

            dz2 = np.dot(W3.T, dz3) * (1 - a2**2)
            dW2 = 1./m * np.dot(dz2, a1.T) + (lambd/(m*2))*W2
            db2 = 1./m * np.sum(dz2, axis=1, keepdims = True)

            dz1 = np.dot(W2.T, dz2) * (1 - a1**2)
            dW1 = 1./m * np.dot(dz1, X.T) + (lambd/(m*2))*W1
            db1 = 1./m * np.sum(dz1, axis=1, keepdims = True)

            gradients = {"dz3": dz3, "dW3": dW3, "db3": db3,"da2": a2,
                         "dz2": dz2, "dW2": dW2, "db2": db2,"da1": a1,
                         "dz1": dz1, "dW1": dW1, "db1": db1}
            return gradients
```

```
In [10]: X_assess, Y_assess, cache = backward_propagation_with_regularization_test_case()
         grads = backward_propagation_with_regularization(X_assess, Y_assess, cache, lambd = 0.1)
         print ("dW1 = "+ str(grads["dW1"]))
         print ("dW2 = "+ str(grads["dW2"]))
         print ("dW3 = "+ str(grads["dW3"]))
         dW1 = [[-0.25604646 0.12419418 0.23789701]
         [ 0.31686832 -0.26022835 -0.44829085]]
         dW2 = [[-0.79592188 -0.16316055 0.73723527]
         [-0.81532281 -0.10053272 0.10099491]]
         dW3 = [[-0.27403627 0.06314963 0.26788808]]
```

Expected Output:

dW1 = [[-0.25604646 0.12419418 0.23789701]
[0.31686832 -0.26022835 -0.44829085]]
dW2 = [[-0.79592188 -0.16316055 0.73723527]
[-0.81532281 -0.10053272 0.10099491]]
dW3 = [[-0.27403627 0.06314963 0.26788808]]

Let's now run the model with L2 regularization ($\lambda = 0.7$). The model is trained with dev set.

```
In [11]: # Train the model.
         parameters = model(train_X, train_Y, lambda = 0.7, initialization = "he", is_plot=Tr
         ue)
         print("On training set:")
         predictions_train = predict(train_X, train_Y, parameters)
         print("On test set:")
         predictions_test = predict(test_X, test_Y, parameters)
         Cost after iteration 0: 0.6974484493131264
         Cost after iteration 10000: 0.2684918873282239
         Cost after iteration 20000: 0.2268455926201365
```

```
         On this regularized model, the decision boundary is much better than the previous or without regularization.
         On training set:
         Accuracy: 0.938388625592
         On test set:
         Accuracy: 0.93
```

Congrats, the test set accuracy increased to 93. You have beaten the French football team!

You have outperformed the learning state-of-the-art, and you had the decision boundary.

```
In [12]: plt.title("Model with L2-regularization")
         plt.xlabel("X1")
         plt.ylabel("X2")
         plt.scatter(X[0, :], X[1, :], c=np.squeeze(Y), s=40, cmap=plt.cm.Spectral)
         plt.show(ax=ax, block=False)
```


Observations:

- The value of λ is a hyperparameter that you can look using a dev set.
- It looks like your regularized model (with $\lambda = 0.7$) to the data is also possible to "overfitted", resulting in a model with high bias.

What's all hyperparameters actually doing?

L2 regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It makes the cost function smoother and it may lead to less flexible fitting. This leads to smoother model, which in this case is better fitting.

What else should remember -- the implications of L2-regularization on:

- The cost computation:
  - A regularization term is added to the cost
- The backpropagation function:
  - There are extra terms for the gradients with respect to weight matrices
- Weights end up smaller ("weight decay"):
  - Weights are pushed to smaller values.

3 - Dropout

Finally, **dropout** is a widely used regularization technique that is specific to Deep Learning. **It randomly shuts down some neurons in each iteration.** Watch Hinton this course to get a feel for what this means.

Figure 2 : Drop out on the second hidden layer

At each iteration, you shut down (set to zero) neurons with probability 0.5, and then you shut down some neurons again in the next iteration. We forward propagate twice (with a different mask) for each training example. Each iteration, you'll train a slightly different model that considers only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time.

3.1 - Forward propagation with dropout

Exercise: Implement the forward propagation with dropout. You are using a 3 layer neural network, and will add dropout to the first and second hidden layers. We do not apply dropout to the input layer or output layer.

Instructions: You should first do two steps for the first and second layers. To the first, you are going to carry out 3 steps:

1. Two variables, $D^{[1]}$ and $D^{[2]}$, will store a random matrix of values between 0 and 1. To do this, you are going to carry out 3 steps:
   a. You are using vectorization, so you want an array of dimension $(n^{[1]}, m)$ to be random, where $n^{[1]}$ is the number of units in layer 1, and m is the number of examples. So you'll write something like $D^{[1]} = np.random.rand(n^{[1]}, m)$
   b. Set each entry of $D^{[1]}$ to 0 with probability (1.keep_prob) or 1 with probability (keep_prob), by thresholding values in $D^{[1]}$ appropriately. Hint: to find out which values in a matrix X are smaller than the threshold, you can do: `X = (X < threshold)`. (Keep at mind that `A = np.random.rand(..., ...)` and then `X = (X < threshold)` will initialize first a matrix of shape `A.shape` with random values, and then compare each of its elements with the threshold.)
   c. Set each entry of $D^{[1]}$ to 0 or 1 (using the result of step b)
2. You'll then shut some neurons. You do this by also zeroing-out the rows of the cost with both the same neurons shut as without drop out. (This technique is also called Inverted dropout).

```
In [13]: # GRADED FUNCTION: forward_propagation_with_dropout

         def forward_propagation_with_dropout(X, parameters, keep_prob = 1.0):
             """
             Implements the forward propagation: LINEAR -> RELU + DROPOUT -> LINEAR -> RELU + DROPOUT -> LINEAR -> SIGMOID.

             Arguments:
             X -- input dataset, of shape (2, number of examples)
             parameters -- python dictionary containing your parameters "W1", "b1", "W2", "b2", "W3", "b3":
                             W1 -- weight matrix of shape (20, 2)
                             b1 -- bias vector of shape (20, 1)
                             W2 -- weight matrix of shape (3, 20)
                             b2 -- bias vector of shape (3, 1)
                             W3 -- weight matrix of shape (1, 3)
                             b3 -- bias vector of shape (1, 1)
             keep_prob - probability of keeping a neuron active during drop-out, scalar
             """
             np.random.seed(1)

             # retrieve parameters
             W1 = parameters["W1"]
             b1 = parameters["b1"]
             W2 = parameters["W2"]
             b2 = parameters["b2"]
             W3 = parameters["W3"]
             b3 = parameters["b3"]

             # LINEAR -> RELU + DROPOUT -> LINEAR -> RELU + DROPOUT -> LINEAR -> SIGMOID
             Z1 = np.dot(W1, X) + b1
             A1 = np.maximum(0,Z1)
             ### START CODE HERE ### (not always the same as the paper, see above)
             D1 = np.random.rand(A1.shape[0],A1.shape[1])  # Step 1: initialize matrix D1 = np.random.rand(..,..)
             D1 = (D1 < keep_prob)  # Step 2: convert entries of D1 to 0 or 1 (using keep_prob as th
             ### END CODE HERE ###
             A1 = A1 * D1  # Step 3: shut down some neurons of A1
             A1 = A1 / keep_prob  # Step 4: scale the value of neurons that haven't been shut down
             Z2 = np.dot(W2, A1) + b2
             A2 = np.maximum(0,Z2)
             ### START CODE HERE ### (not always the same as the paper, see above)
             D2 = np.random.rand(A2.shape[0],A2.shape[1])  # Step 1: initialize matrix D2 = np.random.rand(..,..)
             D2 = (D2 < keep_prob)  # Step 2: convert entries of D2 to 0 or 1 (using keep_prob as th
             ### END CODE HERE ###
             A2 = A2 * D2  # Step 3: shut down some neurons of A2
             A2 = A2 / keep_prob  # Step 4: scale the value of neurons that haven't been shut down
             Z3 = np.dot(W3, A2) + b3
             A3 = sigmoid(Z3)

             cache = (Z1, D1, A1, W1, b1, Z2, D2, A2, W2, b2, Z3, A3, W3, b3)

             return A3, cache
```

```
In [14]: X_assess, parameters = forward_propagation_with_dropout_test_case()
         print("keep_prob = 0.7 -> activations[1] = [[")
         print(str(A3[0,:]).replace("[","").replace("]",""))
         keep_prob = 0.7 -> activations[1] = [[
         0.36974721 0.00305176 0.04565099 0.49683389 0.62341006
```