Improving Deep Neural Networks:
Hyperparameter tuning, Regularization and Optimization

Week-1

In the 1st course, we have implemented a Deep Neural Network and got the intuition of how forward propagation and backward propagation works, here we would like to improve their efficiency by tuning the hyperparameters and improving the problem of over-fitting (High Variance) and under-fitting (High Bias), in more simpler words, make the neural network more optimised.

In week 1, we start with understanding the importance of dividing the data as Training/Dev/Test set as it is the best way to evaluate our neural networks performance.

Earlier, the data sets were divided as:

BEFORE BIG DATA: Training 60%	Dev 20%	Test 20%

Now, the data sets are divided as:

AFTER BIG DATA: Training 98%	Dev 1%	Test 1%

With Big Data coming in, we have petabytes of data and our data set from 1000's of examples have increased to millions of examples but the concept of dividing data into Train/Dev/Test set remains same but only their fraction in the total changed.

* One important consideration to take note is that, both Training and Test set must come from same distribution.

Example: Training Set -> DSLR Cameras
Test Set -> Mobile Cameras
} Both the data needs to come from either DSLR or cameras.

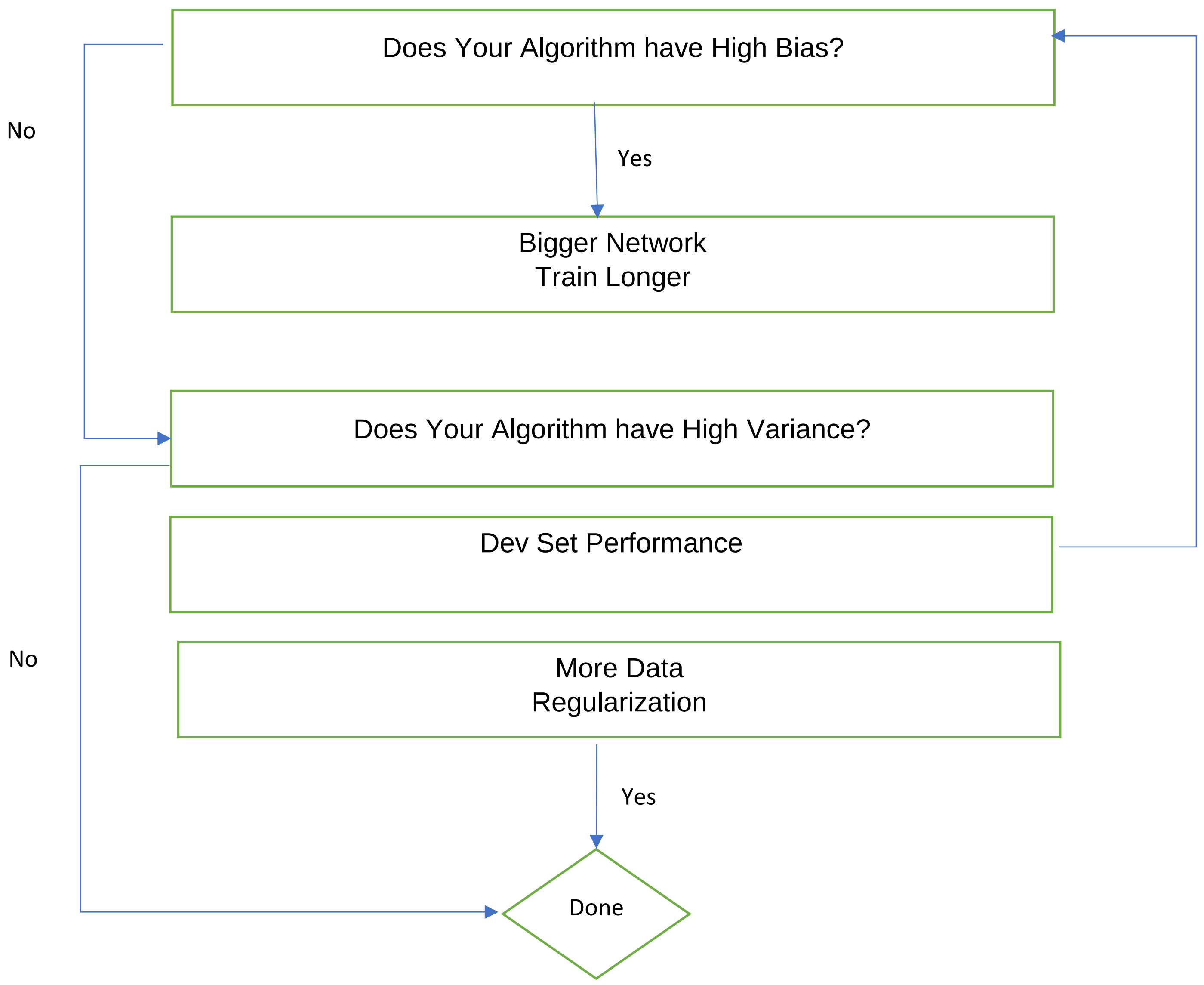
We have seen how to identify whether a learning algorithm is suffering from High Bias (Under-Fitting) or High Variance (Over-Fitting)?

Training Set -> 1.1 % Test Set -> 11%	Over-Fitting High Variance	Training Set -> 15 % Test Set -> 30%	High Bias & High Variance
Training Set -> 15 % Test Set -> 16%	Under-Fitting High Bias	Training Set -> 0.5 % Test Set -> 1%	Low Bias & Low Variance

We have seen, how to solve the problem of over-fitting and under-fitting as follows:

1. High Bias (Under-Fitting)
   - Bigger Network
   - Train Longer
2. High Variance (Over-Fitting)
   - More Data
   - Regularization

Basic recipe of a Machine Learning problem would be as follows:



We understood the need for regularisation, we regularise the features so that they don't overfit the training data and perform well both on Training set and Test Set.

$$min_{w,b}\, J(w,b) \;=\frac{1}{m}\sum_{i=1}^{m}(y,\hat{y})+\frac{\lambda}{2m}||w||^2$$

So, for a given λ (Regularisation -parameter), we want to minimise this equation, for which our individual feature vector needs to be minimal, so it tries to make this term $\frac{\lambda}{2m}||w||^2$ to '0' which means $w\approx 0$, which leads to a High Bias (under-fitting) condition, we need to select a λ in such a way that our training set does not go from a situation of High Variance to High Bias, we need to select λ such that it's almost a perfect fit.

Next, we learnt about Dropout Regularisation wherein we randomly make certain input features to'0' so that our algorithm doesn't over-depend on few particular features.

Other regularisation methods included Data augmentation and Early Stopping.

In Data augmentation, we tried to make more dummy data from the existing data which means the size of our Training set increases which significantly reduces the problem of over-fitting.

Early stopping, as the name suggests, we should stop our algorithm after certain number of iterations before it overfits our training set.

After learning about Regularisation, we started seeing other optimization parameters:

We started with Normalizing inputs so that we treat all of them equally.

Your input features can't be something like this:	Your input features should be something like this:
X1: 0 - 1	X1: 0-1
X2: 100 - 200	X2: -1 - 1
X3: -100- 5	X3: 1- 2

$$X \coloneqq X - \mu$$
$$X \mathrel{/}= \sigma^2$$

By doing this, we can change the shape of the contour from ellipse into circle which reduces the problem of over-shooting.

We saw that we need the initialise the weights in such a way that they don't vanish (or) explode and used XAVIER Initialisation to initialise the weights so that they don't vanish (or) explode.

We have also seen, how the very definition of derivative can help us in debugging, to know which part of the algorithm is not working.

$$d\theta_{approx} = \frac{J(\theta_1, \theta_2, \theta_3 \dots\dots\dots + \varepsilon) - J(\theta_1, \theta_2, \theta_3 \dots\dots\dots \theta_n)}{2\varepsilon}$$

$$d\theta = \frac{dJ}{d\theta_i}$$

Is $d\theta_{approx} \approx d\theta$?

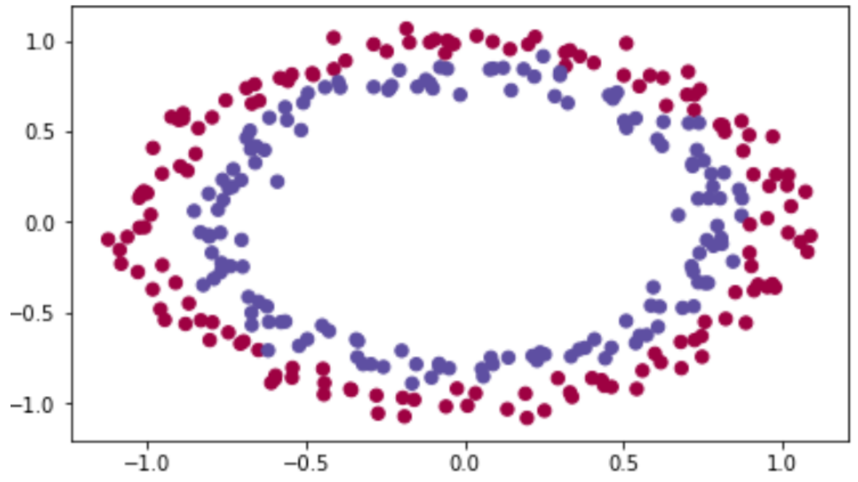
$$Check\ , \frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} \approx \boxed{\begin{array}{l} 10^{-7} - \ Great \\ 10^{-5} - \ OK \\ 10^{-3} - \ Wrong \end{array}}$$

You can check individual components like $db^{[l]}$ and $dW^{[l]}$, to see which part of the algorithm is wrong.

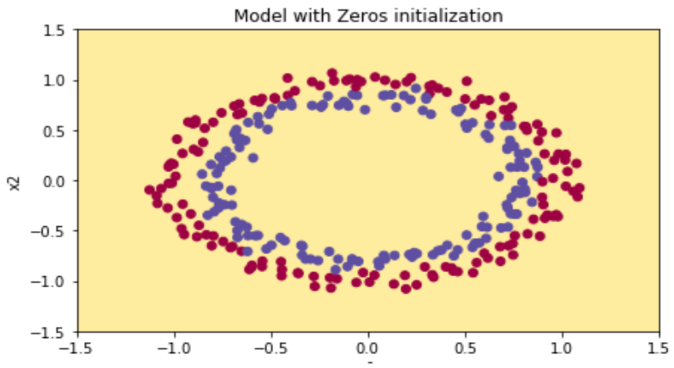
Programming Assignment 1 (Initialisation):

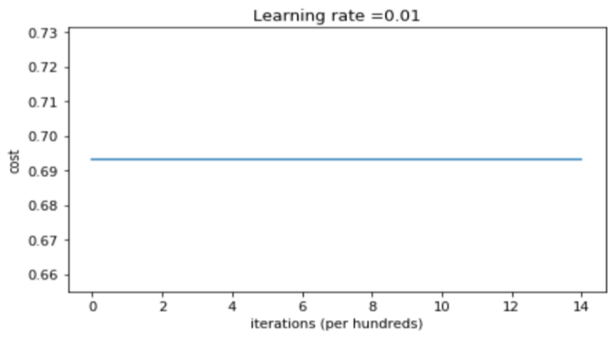
Problem Statement:

We should make a decision boundary separating Red and Blue dots:



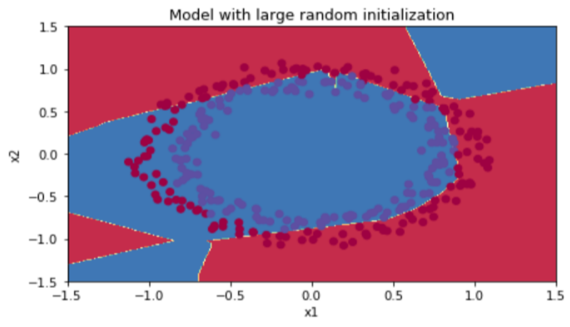
We Initially trained the algorithm with the 3-layer Neural Network we made in the last course but since we initialised all the weights to zeros, so we ended with all zeros and an accuracy of 50%.

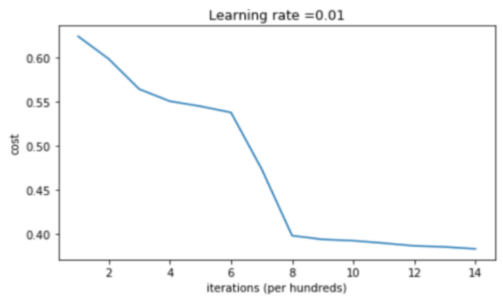




Then, we initialised the all the weights to large numbers. We Observe:

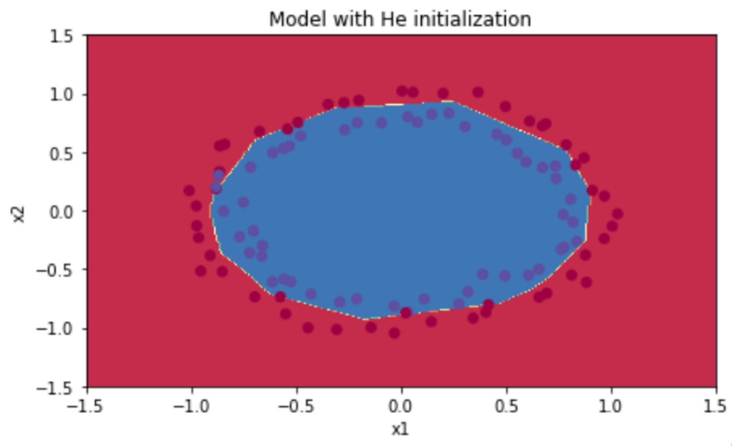
1. The cost starts very high. This is because with large random-valued weights, the last activation (sigmoid) outputs results that are very close to 0 or 1 for some examples, and when it gets that example wrong it incurs a very high loss for that example. Indeed, when $(\log(a^{[3]}) = \log(0))$, the loss goes to infinity.
2. Poor initialization can lead to vanishing/exploding gradients, which also slows down the optimization algorithm.
3. If you train this network longer you will see better results, but initializing with overly large random numbers slows down the optimization.

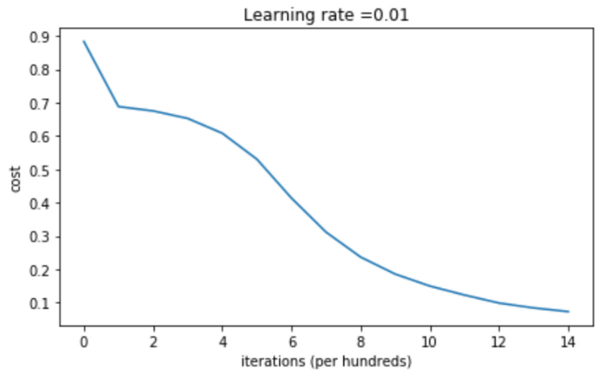




Then, we tried with the He Initialisation (name of the scientist He et al), similar to Xavier Initialisation.

$$W^{[l]} *= \sqrt{\frac{2}{n^{[l-1]}}}$$





Results:

With Zero Initialisation
Output – All 0's
Training set – 50 %
Test Set -50%
Comment: Fails to break Symmetry

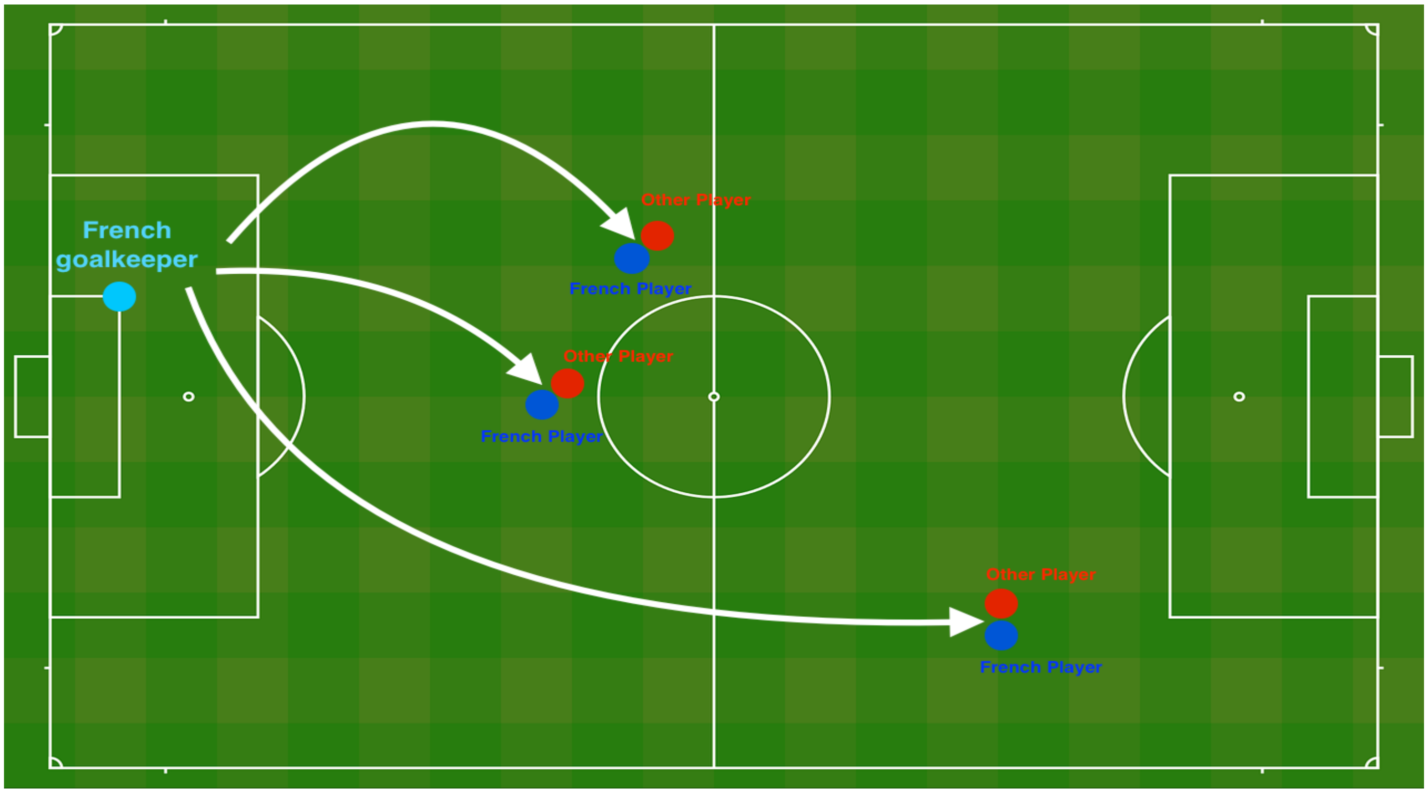
With Large Random Initialisation
Training set – 83 %
Test Set -50%
Comment: Too large weights

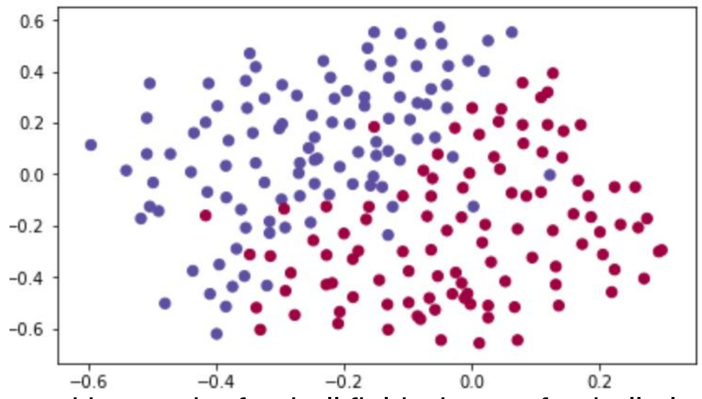
With He Initialisation
Training set – 99.3 %
Test Set -96%
Comment: Recommended Method

Programming Assignment 2 (Regularisation):

Problem Statement:

You have just been hired as an AI expert by the French Football Corporation. They would like you to recommend positions where France's goal keeper should kick the ball so that the French team's players can then hit it with their head.





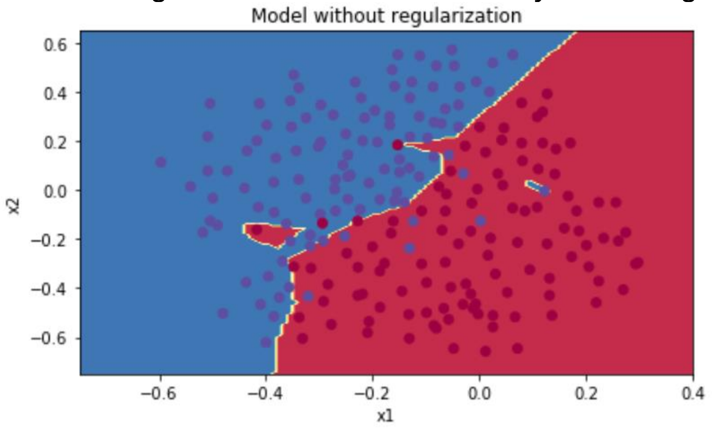
Each dot corresponds to a position on the football field where a football player has hit the ball with his/her head after the French goal keeper has shot the ball from the left side of the football field.

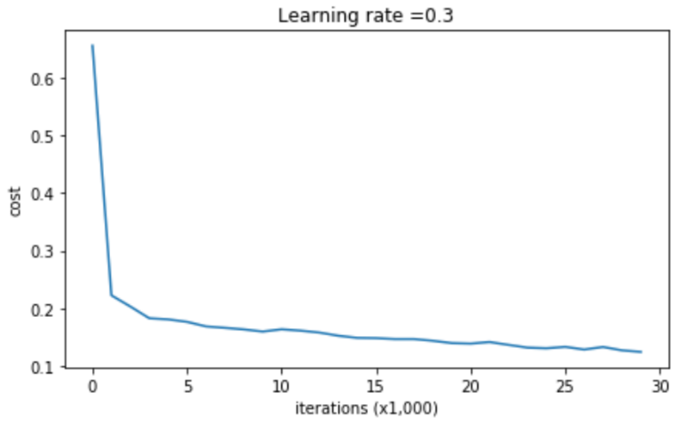
- If the dot is blue, it means the French player managed to hit the ball with his/her head
- If the dot is red, it means the other team's player hit the ball with their head

Goal: Use a deep learning model to find the positions on the field where the goalkeeper should kick the ball.

Before understanding the importance of regularisation, we 1st tried to implement a non-regularised model to understand the need for regularisation:

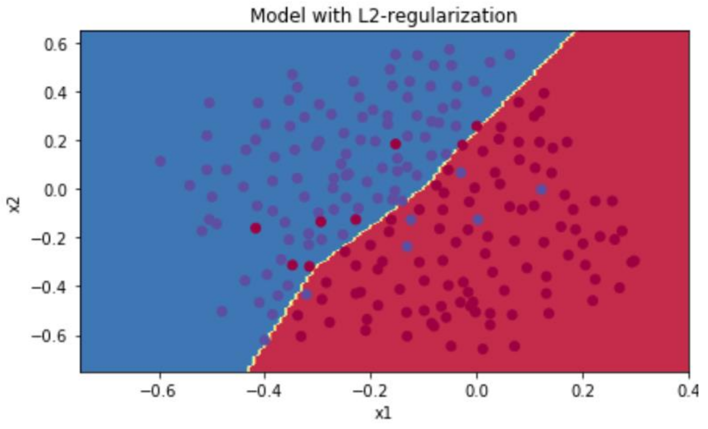
The train accuracy is 94.8% while the test accuracy is 91.5%. This is the **baseline model** (you will observe the impact of regularization on this model).
The non-regularized model is obviously overfitting the training set. It is fitting the noisy points!

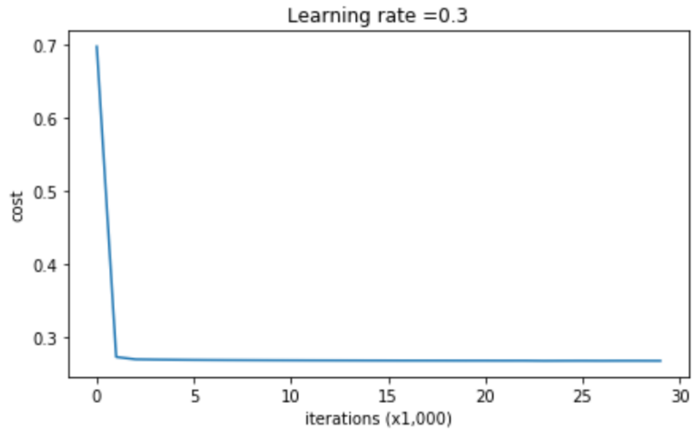




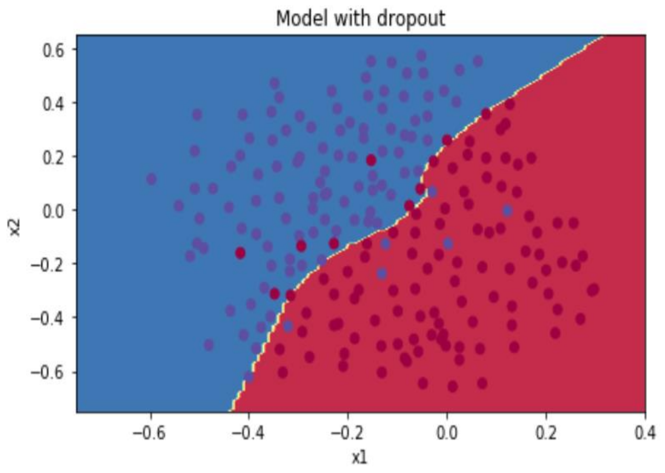
Then, we implemented a L2 regularisation model

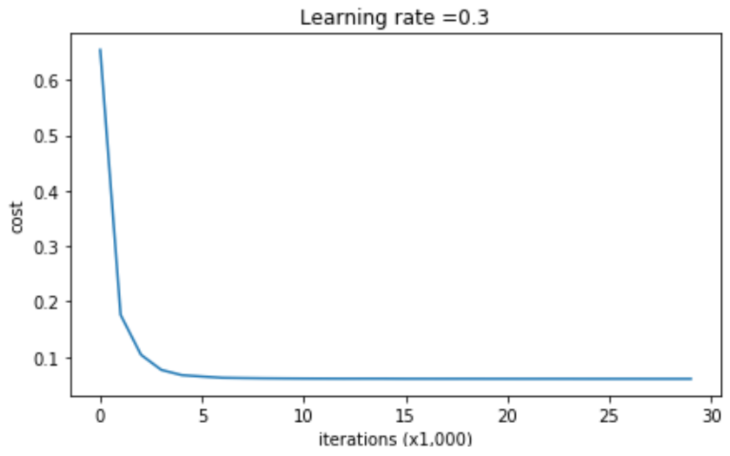
$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} * \log(a^{[L](i)}) + (1 - y^{(i)}) * \log(1 - a^{[L](i)})\right)}_{\text{Cross- Entropy Cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j} W_{kj}^{[l]2}}_{\text{L-2 Regularisation Cost}}$$





Then we implemented, the dropout regularisation.
Dropout works great! The test accuracy has increased again (to 95%).
Our model is not overfitting the training set (92.8%) and does a great job on the test set.





Observations:

1. The value of λ is a hyperparameter that you can tune using a dev set.
2. L2 regularization makes your decision boundary smoother. If λ is too large, it is also possible to "oversmooth", resulting in a model with high bias.
3. **What is L2-regularization 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 becomes too costly for the cost to have large weights!. This leads to a smoother model in which the output changes more slowly as the input changes.
4. A **common mistake** when using dropout is to use it both in training and testing.
   You should use dropout (randomly eliminate nodes) only in training.
   Deep learning frameworks like tensorflow, PaddlePaddle, keras or caffe come with a dropout layer implementation.

Results:

Without Regularisation:
Training set – 94.7 %
Test Set - 91.5 %
Comment: Over-fitting the training set

With L2 Regularisation:
Training set – 93.8 %
Test Set -93%
Comment: Not Over-fitting the data anymore

With Dropout Regularisation:
Training set – 92.8 %
Test Set - 95%
Comment: Does magic in increasing the test set performance.

1. Regularization will help you reduce overfitting.
2. Regularization will drive your weights to lower values.
3. L2 regularization and Dropout are two very effective regularization techniques.
4. Note that regularization hurts training set performance! This is because it limits the ability of the network to overfit to the training set.

Programming Assignment 3 (Gradient Checking):

Problem Statement:

You are part of a team working to make mobile payments available globally, and are asked to build a deep learning model to detect fraud--whenever someone makes a payment, you want to see if the payment might be fraudulent, such as if the user's account has been taken over by a hacker.

But backpropagation is quite challenging to implement, and sometimes has bugs. Because this is a mission-critical application, your company's CEO wants to be really certain that your implementation of backpropagation is correct. Your CEO says, "Give me a proof that your backpropagation is actually working!" To give this reassurance, you are going to use "gradient checking".

Observations:

1. It seems that there were errors in the backward_propagation code.
   Good that we've implemented the gradient check.

```
def backward_propagation_n(X, Y, cache):
    """
    Implement the backward propagation.

    Arguments:
    X -- input datapoint, of shape (input size, 1)
    Y -- true "label"
    cache -- cache output from forward_propagation_n()

    Returns:
    gradients -- A dictionary with the gradients of the cost with respect to each
parameter, activation and pre-activation variables.
    """

    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)
    db3 = 1./m * np.sum(dZ3, axis=1, keepdims = True)

    dA2 = np.dot(W3.T, dZ3)
    dZ2 = np.multiply(dA2, np.int64(A2 > 0))
    dW2 = 1./m * np.dot(dZ2, A1.T) * 2
    db2 = 1./m * np.sum(dZ2, axis=1, keepdims = True)

    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, np.int64(A1 > 0))
    dW1 = 1./m * np.dot(dZ1, X.T)
    db1 = 4./m * np.sum(dZ1, axis=1, keepdims = True)

    gradients = {"dZ3": dZ3, "dW3": dW3, "db3": db3,
                 "dA2": dA2, "dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dA1": dA1, "dZ1": dZ1, "dW1": dW1, "db1": db1}

    return gradients
```

2. Gradient Checking is slow! Approximating the gradient with $\frac{dJ}{d\theta} \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}$ is computationally costly. For this reason, we don't run gradient checking at every iteration during training. Just a few times to check if the gradient is correct.

3. Gradient Checking, at least as we've presented it, doesn't work with dropout. You would usually run the gradient check algorithm without dropout to make sure your backprop is correct, then add dropout.

Results:

1. Gradient checking verifies closeness between the gradients from backpropagation and the numerical approximation of the gradient (computed using forward propagation).
2. Gradient checking is slow, so we don't run it in every iteration of training. You would usually run it only to make sure your code is correct, then turn it off and use backprop for the actual learning process.