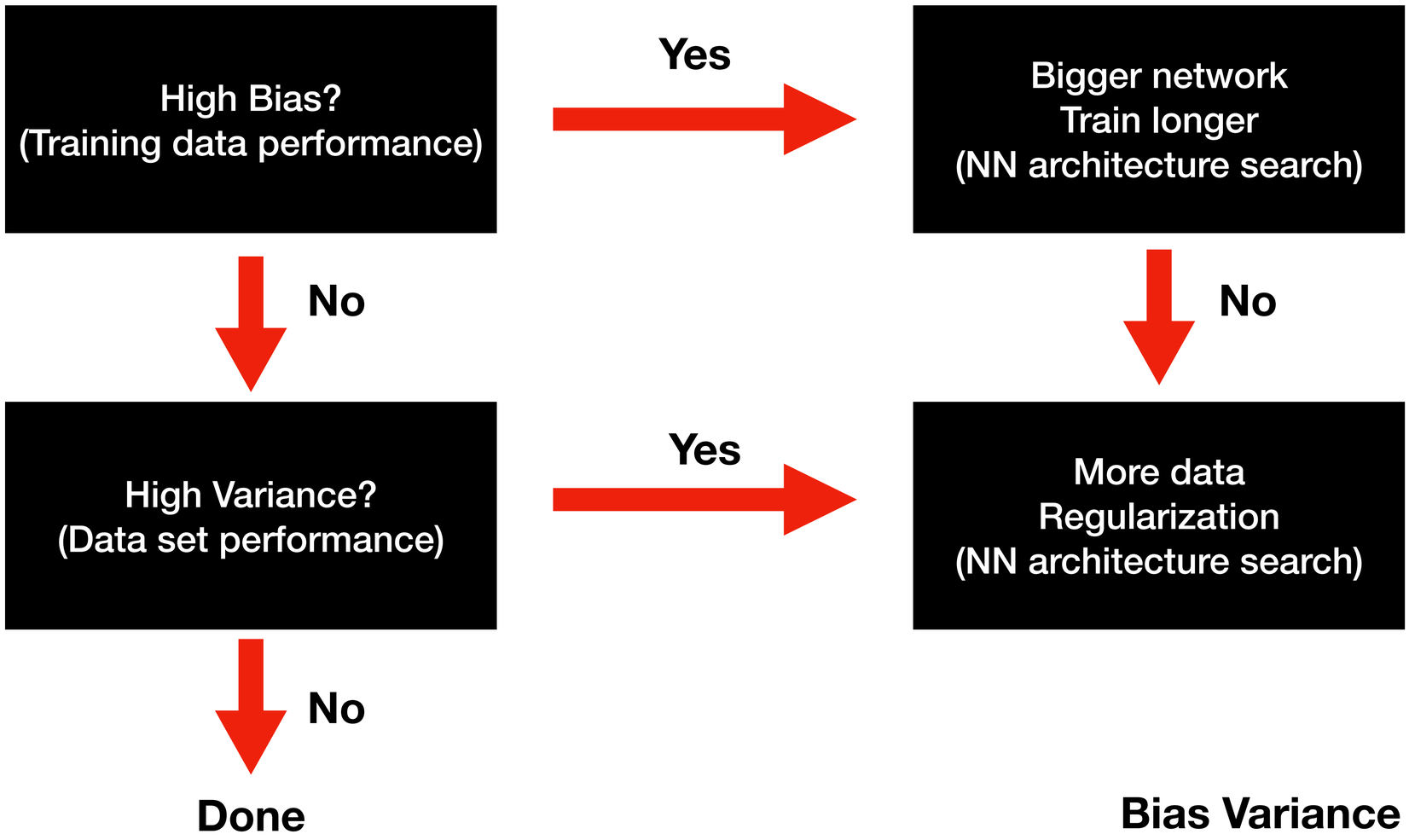
Regularization

박달님 06.18.21

Basic Instruction for Machine Learning



Bias Variance tradeoff

Norm

- 벡터의 크기를 측정하는 방법/두 벡터 사이의 거리를 측정하는 방법
- p는 Norm의 차수를 의미. p=1 -> L1 norm, p=2 -> L2 norm
- n = 해당 벡터의 원소 수

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

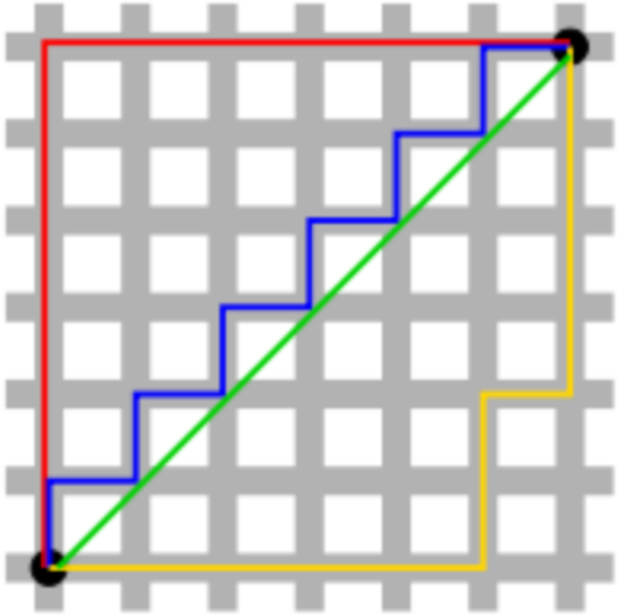
L1 Norm

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|, \text{ where } (\mathbf{p}, \mathbf{q}) \text{ are vectors } \mathbf{p} = (p_1, p_2, \ldots, p_n) \text{ and } \mathbf{q} = (q_1, q_2, \ldots, q_n)$$

- L1 Norm: 벡터 p, q의 각 원소들의 차이의 절대값의 합

L1 Norm vs. L2 Norm

- 두 점 사이의 거리에서 L1 Norm은 빨간색, 파란색, 노란색 선으로 표현 -> many paths
- L2 Norm은 초록색 선으로만 표현 -> Unique shortest path



L1 Loss vs. L2 Loss

- L1 Loss: 실제 값과 예측치 사이의 오차 값의 절대값을 구하고 그 오차들의 합으로 정의. (Least Absolute Errors LAE)

$$L = \sum_{i=1}^{n} |y_i - f(x_i)|$$

- L2 Loss: 오차의 제곱의 합으로 정의. (Least Sqaures Error LSE)

$$L = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

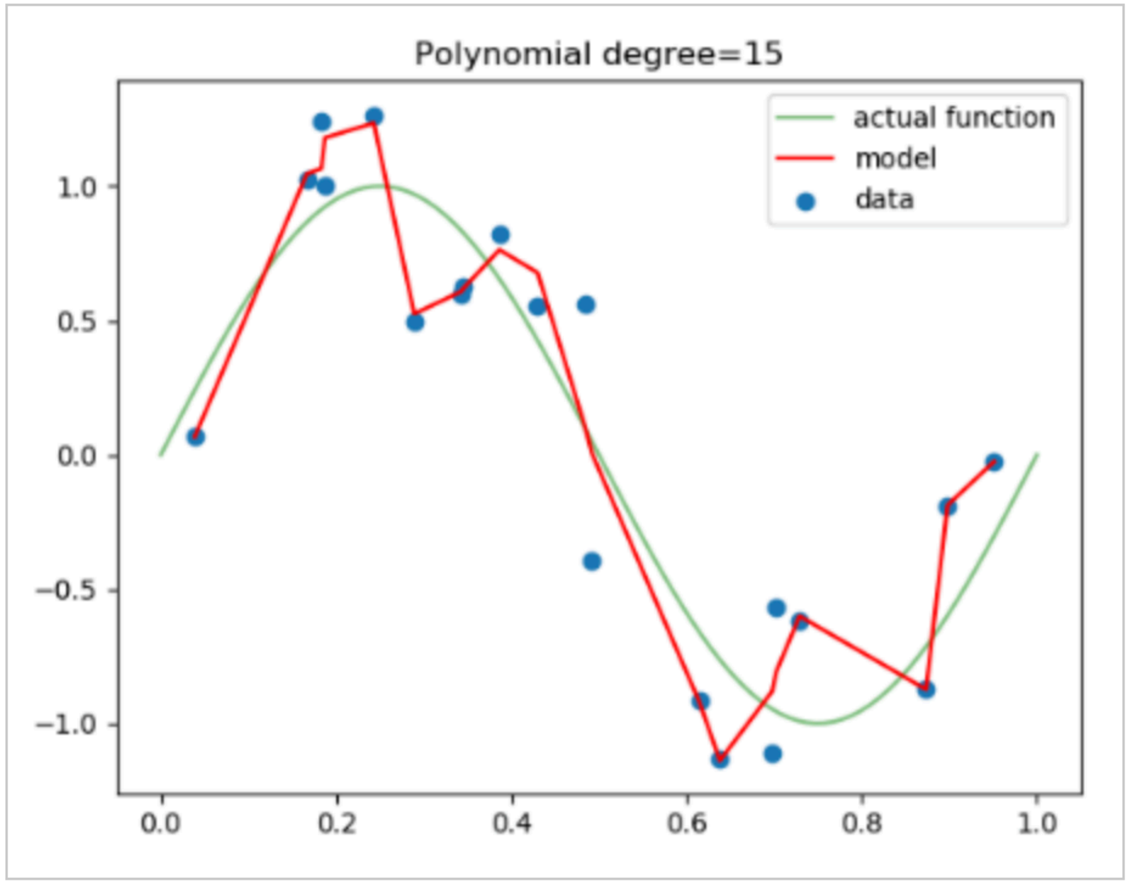
- L2는 직관적으로 오차의 제곱을 더함. Outlier의 더 큰 영향을 받음.
- Outlier가 적당히 무시되어야 하는 경우: L1 loss 사용

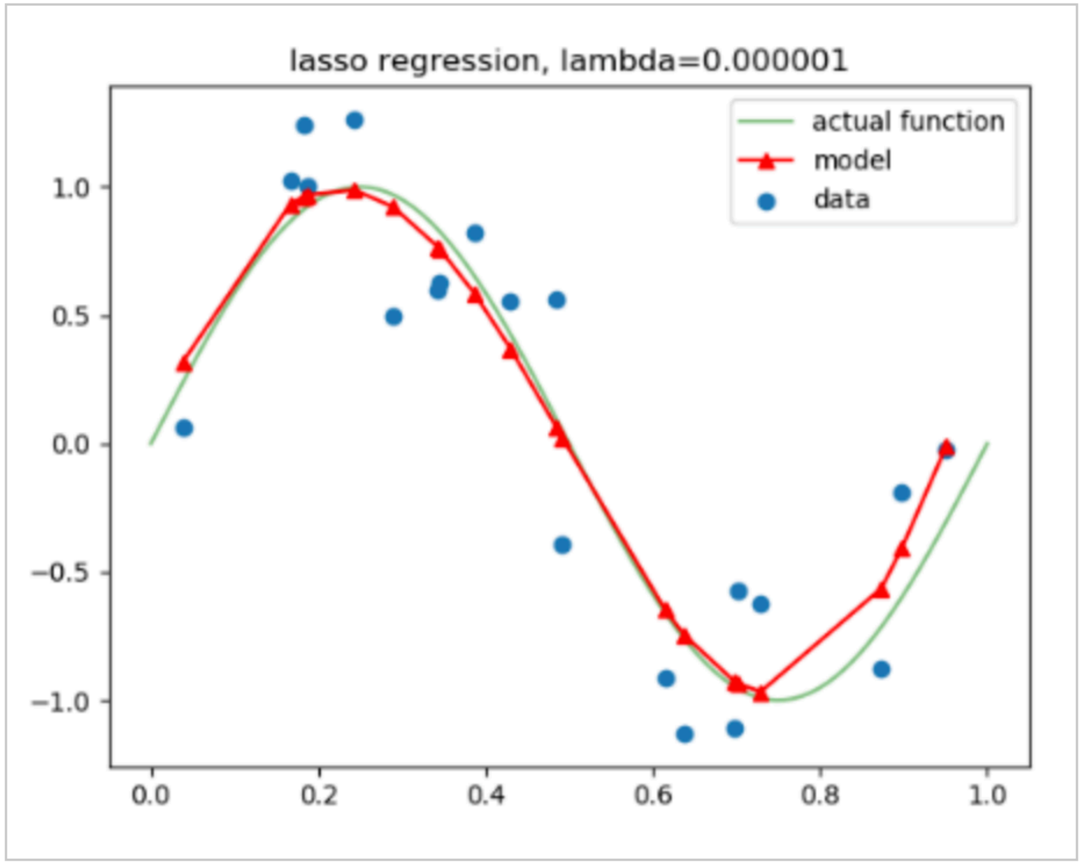
Regularization

- 모델 복잡도에 대한 패널티로 Overfitting을 예방하고 generalization (일반화) 성능을 높이는데 도움을 준다.

- Regularization 방법: L1 Regularization, L2 Regularization, Dropout, Early stopping, etc.

- 모델을 쉽게 만드는 방법은 단순하게 cost function 값이 작아지는 방향으로만 진행하는것. 그러나 이럴 경우 **특정 가중치가 너무 큰 값**을 가지기 때문에 **모델의 일반화 성능이 떨어지게** 된다.





- 특정 가중치가 너무 과도하게 커지지 않도록 하여 과적합을 막아줌.

L1 Regularization

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y_i}) + \frac{\lambda}{2}|w|\}$$

$L(y_i, \hat{y_i})$: 기존의 Cost function

- Cost function에 가중치의 절대값을 더해줌
- 기존의 cost function에 가중치의 크기가 포함되면서 가중치가 너무 크지 않은 방향으로 학습 되도록 함
- 이때 λ는 learning rate 같은 상수로 0에 가까울 수록 정규화의 효과가 없어짐
- L1 regularization을 사용하는 regression model을 least absolute shrinkage and selection operator (Lasso) regression 이라고 함
- L1 regularization을 쓰게 되면 w가 sparse하게 된다 -> w vector 안에 0가 많아진다
- Compress model에 유용: parameter들이 0이면 더 적은 메모리가 필요하다 (많이 쓰이지는 않는다)

L2 Regularization

$$Cost = \frac{1}{n}\sum_{i=1}^{n}\{L(y_i, \hat{y_i}) + \frac{\lambda}{2}|w|^2\}$$

- Cost function 에 가중치의 제곱을 포함하여 더함으로써 L1 Regularization과 마찬가지로 가중치가 너무 크지 않은 방향으로 학습 -> Weight decay
- L2 Regression을 사용하는 Regression model을 Ridge Regression이라고 함
- L1 보다 더 자주 사용함

L1, L2 Regularization 차이, 선택 기준

- Regularization 의 의미: 가중치 w가 작아지도록 학습한다는 것은 결국 local noise 의 영향을 덜 받도록 하겠다는 것, 즉 outlier의 영향을 더 적게 받도록 하겠다는 것

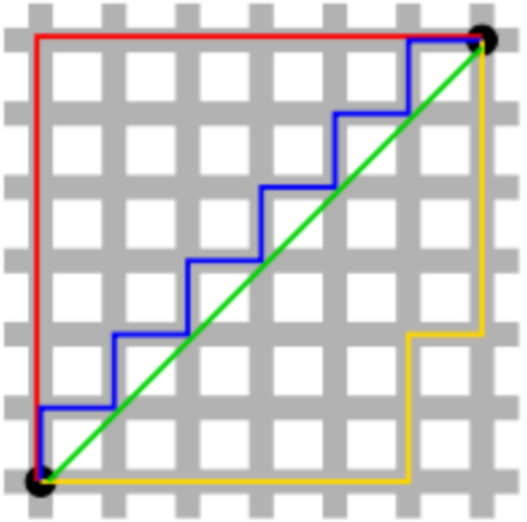
$$a = (0.3, -0.3, 0.4)$$
$$b = (0.5, -0.5, 0)$$

$$||a||_1 = |0.3| + |-0.3| + |0.4| = 1$$
$$||b||_1 = |0.5| + |-0.5| + |0| = 1$$

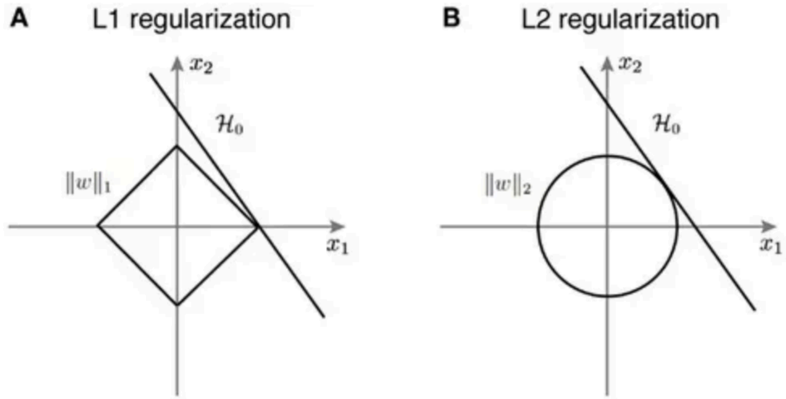
$$||a||_2 = \sqrt{0.3^2 + (-0.3^2) + 0.4^2} = 0.583095$$
$$||b||_2 = \sqrt{0.5^2 + (-0.5^2) + 0^2} = 0.707107$$

- L2 norm은 각각의 벡터에 대해 항상 unique한 값을 내지만, L1 norm은 경우에 따라 특정 feature 없이도 같은 값을 낼 수 있다

L1, L2 Regularization 차이, 선택 기준



- L1 Norm은 파란색 선 대신 빨간색 선을 사용하여 특정 feature를 0으로 처리하는 것이 가능하다
- L1 Norm은 feature selection이 가능하고 이런 특징이 L1 Regularization에 동일하게 적용될 수 있다
- 따라서 L1은 Sparse model에 적합, convex optimization에 유용하게 쓰인다
- 그러나 L1은 미분 불가능한 점이 있기 때문에 gradient-base learning에서는 주의가 필요함

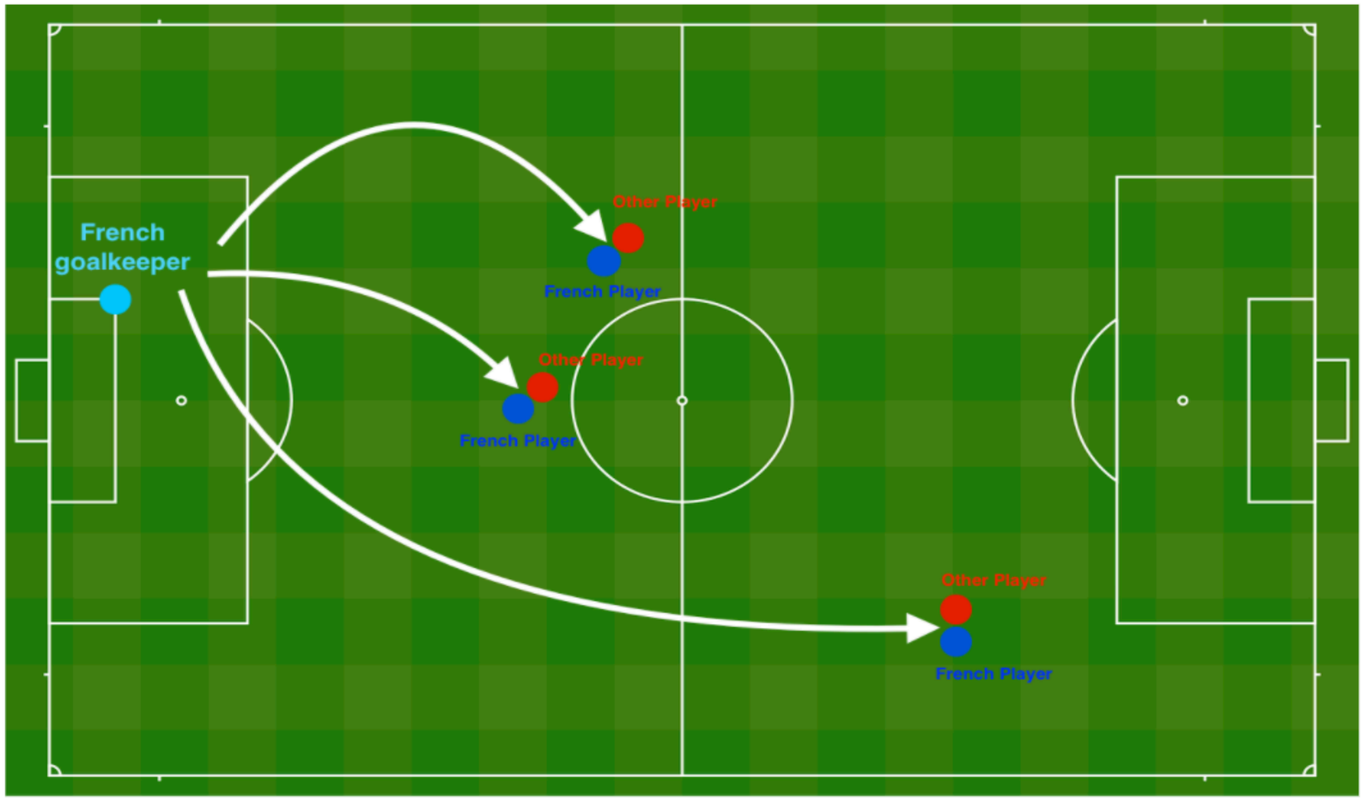


1 - Packages

```
# import packages
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import scipy.io
from reg_utils import sigmoid, relu, plot_decision_boundary, initialize_parameters, load_2D_dataset, predict_dec
from reg_utils import compute_cost, predict, forward_propagation, backward_propagation, update_parameters
from testCases import *
from public_tests import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (7.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2
```

2 - Problem Statement

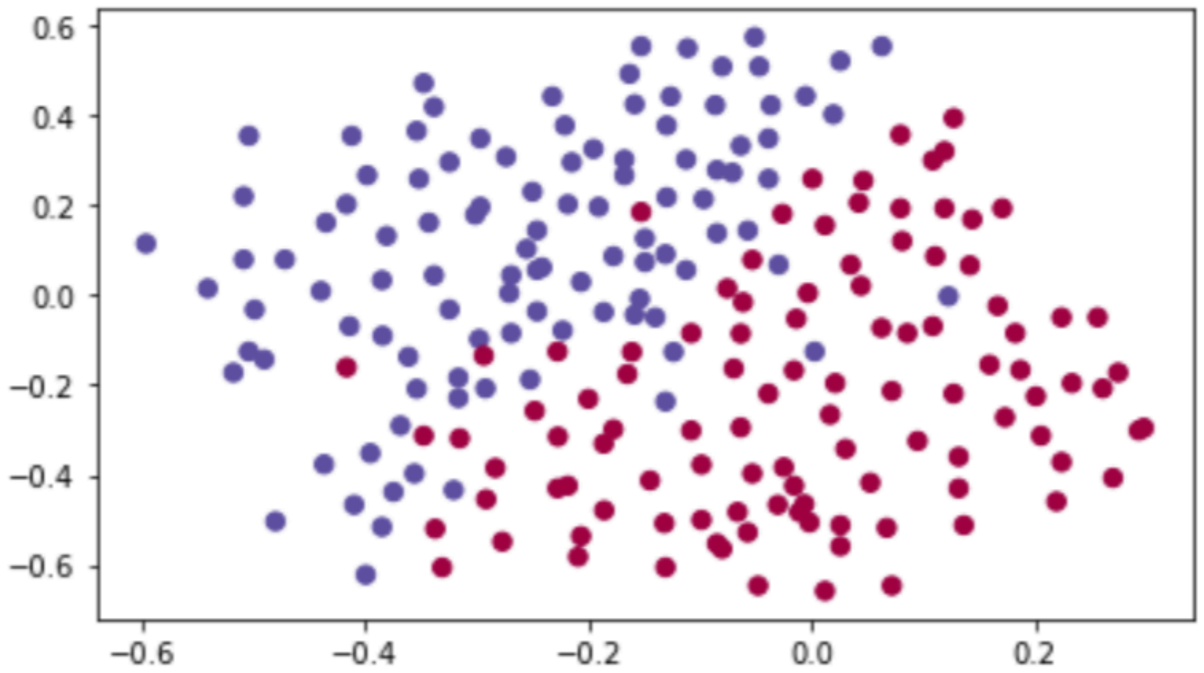
- 프랑스의 골키퍼가 공을 차서 선수들이 머리로 공을 칠 수 있는 위치 구하기



3 - Loading the Dataset

- 각 점은 프랑스 골키퍼가 축구장 왼쪽에서 공을 찬 후 선수가 머리로 공을 쳤던 축구장의 위치에 해당
- 파란색 점 = 프랑스 선수가 공을 머리로 침
- 빨간색 점 = 다른 나라 선수가 공을 머리로 침

```
train_X, train_Y, test_X, test_Y = load_2D_dataset()
```

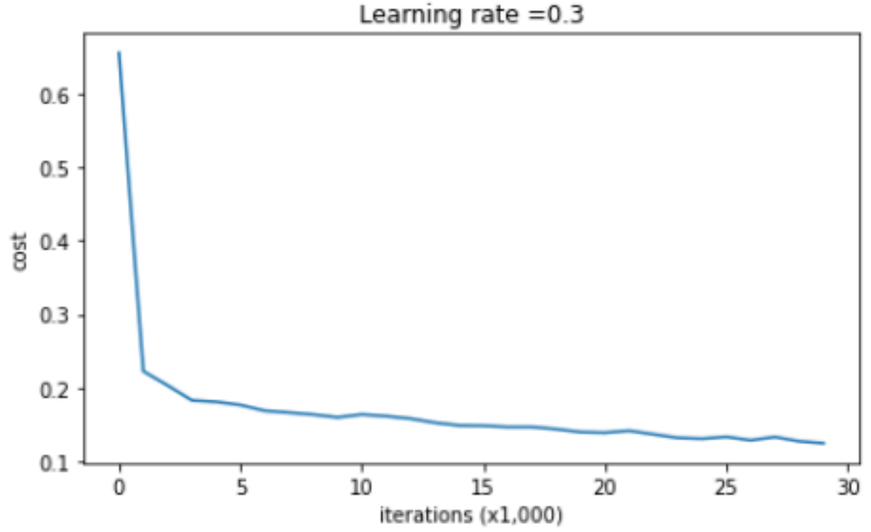


4 - Non-Regularized Model

```
def model(X, Y, learning_rate = 0.3, num_iterations = 30000, print_cost = True, lambd = 0, keep_prob = 1):
    """
    Implements a three-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (input size, number of examples)
    Y -- true "label" vector (1 for blue dot / 0 for red dot), of shape (output size, number of examples)
    learning_rate -- learning rate of the optimization
    num_iterations -- number of iterations of the optimization loop
    print_cost -- If True, print the cost every 10000 iterations
    lambd -- regularization hyperparameter, scalar
    keep_prob - probability of keeping a neuron active during drop-out, scalar.

    Returns:
    parameters -- parameters learned by the model. They can then be used to predict.
    """

    grads = {}
    costs = []                            # to keep track of the cost
    m = X.shape[1]                        # number of examples
    layers_dims = [X.shape[0], 20, 3, 1]

    # Initialize parameters dictionary.
    parameters = initialize_parameters(layers_dims)

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
        if keep_prob == 1:
            a3, cache = forward_propagation(X, parameters)
        elif keep_prob < 1:
            a3, cache = forward_propagation_with_dropout(X, parameters, keep_prob)

        # Cost function
        if lambd == 0:
            cost = compute_cost(a3, Y)
        else:
            cost = compute_cost_with_regularization(a3, Y, parameters, lambd)

        # Backward propagation.
        assert (lambd == 0 or keep_prob == 1)   # it is possible to use both L2 regularization and dropout,
                                                # but this assignment will only explore one at a time
        if lambd == 0 and keep_prob == 1:
            grads = backward_propagation(X, Y, cache)
        elif lambd != 0:
            grads = backward_propagation_with_regularization(X, Y, cache, lambd)
        elif keep_prob < 1:
            grads = backward_propagation_with_dropout(X, Y, cache, keep_prob)

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the loss every 10000 iterations
        if print_cost and i % 10000 == 0:
            print("Cost after iteration {}: {}".format(i, cost))
        if print_cost and i % 1000 == 0:
            costs.append(cost)

    # plot the cost
    plt.plot(costs)
    plt.ylabel('cost')
    plt.xlabel('iterations (x1,000)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

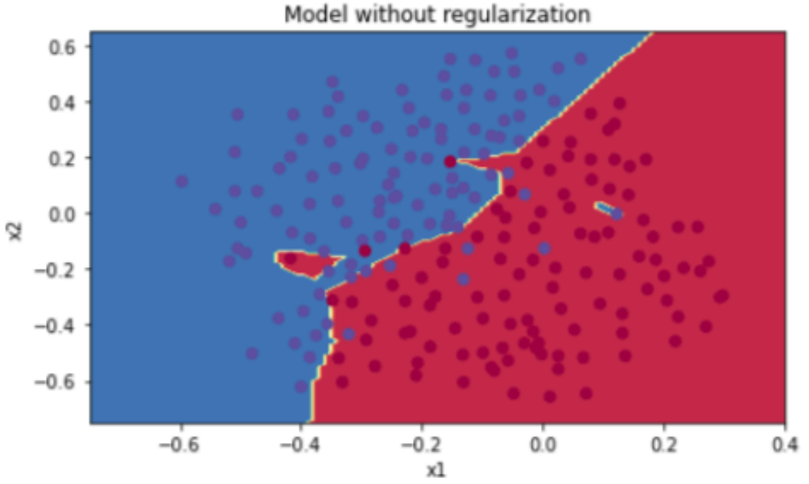
```
parameters = model(train_X, train_Y)
print ("On the training set:")
predictions_train = predict(train_X, train_Y, parameters)
print ("On the test set:")
predictions_test = predict(test_X, test_Y, parameters)
```

```
Cost after iteration 0: 0.6557412523481002
Cost after iteration 10000: 0.16329987525724204
Cost after iteration 20000: 0.13851642423234922
```




```
On the training set:
Accuracy: 0.9478672985781991
On the test set:
Accuracy: 0.915
```

```
plt.title("Model without regularization")
axes = plt.gca()
axes.set_xlim([-0.75,0.40])
axes.set_ylim([-0.75,0.65])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```



5 - L2 Regularization

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

$$J_{regularized} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right) + (1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 regularization cost}}$$

Regularization - Logistic Regression

Neural network over fitting data —> High variance problem -> Regularization

OR get more training data
(but can't always get more data)

Logistic Regression

$$\min_{w,b} J(w,b)$$

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \|w\|_2^2$$

Lambda = regularization parameter

$$\|w\|_2^2 = \sum_{j}^{n} w_j^2 = w^\top w$$

"L2 Regularization"

Euclidean norm "L2 norm"

Don't need to regularize parameter b
W has a lot of parameters, b is just a single number so okay to omit it

Regularization - Neural network

$$J(w^{[1]}, b^{[1]}, \ldots, w^{[L]}, b^{[L]}) = \frac{1}{m} \sum_{i}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \sum_{l}^{L} \|w^{(l)}\|^2$$

$$\|w^{(l)}\|^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$$

W : (n^[1], n^[l-1]) matrix
Number of units in layers [l-1] and layer l

“Frobenius Norm”

Applying gradient descent
Previously: compute dw = (from backpropagation) $\frac{\partial J}{\partial w^l} + \frac{\lambda}{m} w^l$
Then update w^[l] := w^[l] - αdw^[l]
Now add lambda factor

$$w^{[l]} := w^{[l]} - \alpha \left[(\text{from backprop}) + \frac{\lambda}{m} w^{[l]} \right]$$

$$= w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha (\text{from backprop})$$

“weight decay”

5.1 - Compute cost with regularization

```
# 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, of shape (output size, number of examples)
    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

    L2_regularization_cost = lambd * (np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))/(2*m)

    cost = cross_entropy_cost + L2_regularization_cost

    return cost
```

```
A3, t_Y, parameters = compute_cost_with_regularization_test_case()
cost = compute_cost_with_regularization(A3, t_Y, parameters, lambd=0.1)
print("cost = " + str(cost))

compute_cost_with_regularization_test(compute_cost_with_regularization)
```

```
cost = 1.7864859451590758
 All tests passed.
```

5.2 - backward propagation with regularization

```
# GRADED FUNCTION: backward_propagation_with_regularization

def backward_propagation_with_regularization(X, Y, cache, lambd):
    """
    Implements the backward propagation of our baseline model to which we added an L2 regularization.

    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()
    lambd -- regularization hyperparameter, scalar

    Returns:
    gradients -- A dictionary with the gradients with respect to each parameter, activation and pre-activation variable
    """

    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*W3)/m

    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) + (lambd*W2)/m

    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) + (lambd*W1)/m

    db1 = 1. / 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
```

```
t_X, t_Y, cache = backward_propagation_with_regularization_test_case()

grads = backward_propagation_with_regularization(t_X, t_Y, cache, lambd = 0.7)
print ("dW1 = \n"+ str(grads["dW1"]))
print ("dW2 = \n"+ str(grads["dW2"]))
print ("dW3 = \n"+ str(grads["dW3"]))
backward_propagation_with_regularization_test(backward_propagation_with_regularization)
```

```
dW1 =
[[-0.25604646  0.12298827 -0.28297129]
 [-0.17706303  0.34536094 -0.4410571 ]]
dW2 =
[[ 0.79276486  0.85133918]
 [-0.0957219  -0.01720463]
 [-0.13100772 -0.03750433]]
dW3 =
[[-1.77691347 -0.11832879 -0.09397446]]
 All tests passed.
```

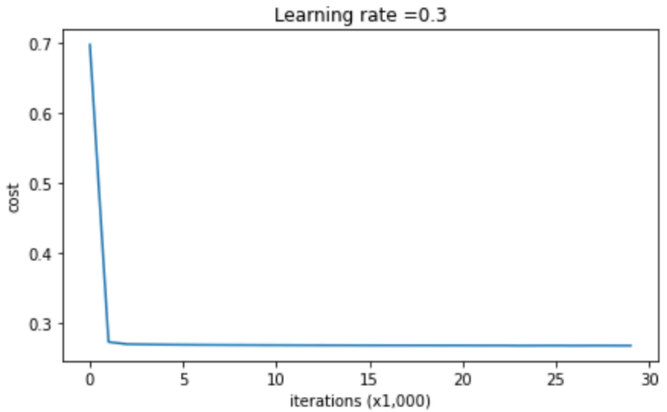
Running the model with L2 regularization (λ = 0.7)

The model() function will call:

- **compute_cost_with_regularization instead of compute_cost**
- **Backward_propagation_with_regularization instead of backward_propagation**

```
parameters = model(train_X, train_Y, lambd = 0.7)
print ("On the train set:")
predictions_train = predict(train_X, train_Y, parameters)
print ("On the 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: 0.26809163371273015
```



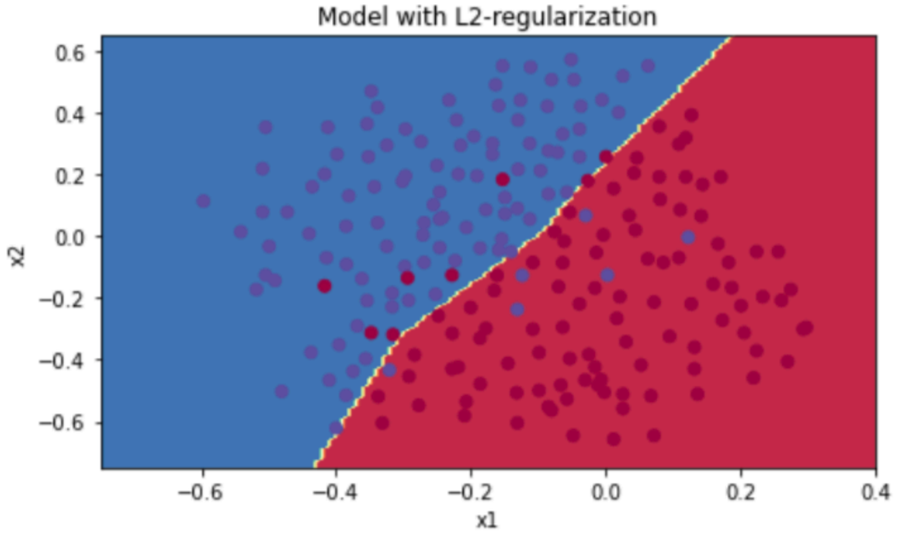

```
On the train set:
Accuracy: 0.9383886255924171
On the test set:
Accuracy: 0.93
```

Test Set Accuracy: 93%, not overfitting the training data anymore

L2 Regularization Conclusion

Decision boundary plot

```
plt.title("Model with L2-regularization")
axes = plt.gca()
axes.set_xlim([-0.75,0.40])
axes.set_ylim([-0.75,0.65])
plot_decision_boundary(lambda x: predict_dec(parameters, x.T), train_X, train_Y)
```



OBSERVATIONS:

- λ는 dev set을 이용해서 조종할 수 있는 hyperparameter 이다
- L2 regularization은 바운더리를 더 smooth하게 해준다
- 만약 λ가 너무 큰 값이면 “oversmooth”가 가능해지며 모델에 high bias가 생긴다
- L2 regularization은 작은 weight를 가지고 있는 모델이 큰 weight를 가지고 있는 모델보다 더 simple 하다는 assumption을 가지고 있다
- 따라서 cost function에서 가중치의 제곱 값에 패널티를 적용하여 모든 가중치를 더 작은 값으로 유도한다
- 이것은 입력이 변함에 따라 출력이 더 느리게 변하는 더 smooth한 모델로 된다