In [1]:
```
import numpy as np
import matplotlib.pyplot as plot
```

Régularisation L2

Mise en situation

Je souhaite améliorer le test sur mon réseau de neurones. J'ai l'impression qu'il y a de l'overfit durant l'entraînement.

Surajustement

On a déjà vu (jours 7 et 8) ce que peuvent être les problèmes de biais, de variance et de surajustement. On va voir comment ça se décline ici.

Définition

Pour rappels:

- Le biais d'un modèle est l'erreur obtenue en entraînement. On n'arrive pas à faire mieux qu'un certain pourcentage d'erreur - c'est notre biais.
- La variance est la différence entre le score à l'entraînement et celui du test. Là, si on arrive à faire sensiblement mieux sur le training, c'est que ce que le modèle reconnaît quelque chose de plus efficace pour lui, mais pas forcément bon pour nous. Il surajuste les données de training.

Bien identifier les problèmes d'overfit

Dans le cadre de notre reconnaissance de chiffres, supposons qu'on trouve un modèle à 96% sur le training et 95% sur le test. Le "1%" d'écart est la variance du modèle. Dans ce cas précis, il n'y a probablement pas de gros problème d'overfit.

Plus exactement, s'il y en a un, il n'est pas urgent de se pencher dessus. On a 4% de biais, et travailler dessus est bien plus intéressant. Arriver à le réduire de moitié (faire 98% donc) peut par voie de conséquence monter le test à 97% peut-être, alors que se focaliser sur la réduction de la variance et la réduire de moitié nous fera atteindre 95.5% !

A l'inverse, notre détecteur de chats faisait, supposons, 90% au training, et 70% au test. Alors oui il y a un sacré biais (10%) mais il y a surtout une grosse variance. Réduire le biais de moitié nous fera faire du 95% au training, et probablement +4% sur le test: réduire la variance de moitié nous fera faire du 80% au test !

Pour éviter de se dispenser, il vaut mieux essayer de régler les problèmes les uns après les autres. Et donc, si le biais pose d'avantage de souci, on se concentrera sur d'autres méthodes en priorité, et si c'est la variance on pourra faire un peu de régularisation, comme on va le voir tout de suite.

Régularisation

On avait vu qu'on pouvait ajouter, à l'entraînement, un certain terme de régularisation, la somme des carrés des θ - et forcer du coup ces valeurs à ne pas devenir trop grandes. Un surajustement s'accompagne souvent d'une grande amplitude dans les coefficients, pour pouvoir mieux coller au problème.

Là, on va tout simplement ajouter la somme des carrés des W. C'est tout. Le terme en plus sera

$$\frac{1}{m}\sum W^2$$

Dérivée

Le coût change, et donc sa dérivée aussi. Mais enfin la c'est pas énorme :

- les gradients des poids b ne sont pas concernés par le changement et leur dérivée ne change donc pas
- Pour les W, on ajoute simplement le terme $\frac{2}{m}W$. Pour un paramètre donné, tous les autres supposés constants, la somme ne porte que sur des constantes (dérivée nulle donc) sauf pour le terme considéré, qui donne $2W$.

Implémentation

In [2]:
```
# Fonctions d'initialisation
def uniform(in_dim, out_dim): return np.random.rand(out_dim, in_dim)
def uniform_100(in_dim, out_dim): return uniform(in_dim, out_dim) * 0.01
def gaussian(in_dim, out_dim): return np.random.randn(out_dim, in_dim)
def xavier(in_dim, out_dim): return gaussian(in_dim, out_dim) / np.sqrt(in_dim)
def he(in_dim, out_dim): return gaussian(in_dim, out_dim) * np.sqrt(2/in_dim)
def bengio(in_dim, out_dim): return gaussian(in_dim, out_dim) / np.sqrt(out_dim + in_dim)

init_functions = {'uniform': uniform,
                  'uniform_100': uniform_100,
                  'gaussian': gaussian,
                  'xavier': xavier,
                  'he': he,
                  'bengio': bengio}

# Les différentes fonctions
def sigmoid(x) : return 1 / (1 + np.exp(-x))
def tanh(x): return (np.exp(x) - np.exp(-x))/(np.exp(x) + np.exp(-x))
def relu(x): return np.maximum(x, 0)

act_functions = {'sigmoid': sigmoid, 'tanh' : tanh, 'relu' : relu}

# Leurs dérivées
def d_sigmoid(x):
    a = sigmoid(x)
    return a * (1 - a)

def d_tanh(x):
    t = tanh(x)
    return 1 - t**2

def d_relu(x):
    return x > 0

act_derivates = {'sigmoid' : d_sigmoid, 'tanh' : d_tanh, 'relu' : d_relu}

# Passe en avant : 1 couche - on utilise le dictionnaire de fonctions
def layer_forward_pass(X, W, b, activation):
    Z = np.dot(W, X) + b
    A = act_functions[activation](Z)
    return Z, A

# Passe en avant : toutes les couches
def model_forward_pass(X, activations, parameters):
    result = {}
    result['A0'] = X
    # Entrée de la première couche: X
    A = X
    for i in range(1, len(activations) + 1):
        # Pour chaque couche, une passe en avant. Les W et b viennent de parameters
        Z_next, A_next = layer_forward_pass(A, parameters['W' + str(i)], parameters['
        result['Z' + str(i)] = Z_next
        result['A' + str(i)] = A_next
        A = A_next
    return result

# Passe en arrière : 1 couche - on utilise le dictionnaire de dérivées
def layer_backward_pass(dA, Z, A_prev, W, activation):
    dZ = dA * act_derivates[activation](Z)
    dW = np.dot(dZ, A_prev.T)
    db = np.sum(dZ, axis=1, keepdims = True)
    dA_prev = np.dot(W.T, dZ)
    return dW, db, dA_prev

# Passe en arrière : toutes les couches
def model_backward_pass(dA_last, parameters, forward_pass_results, activations):
    gradients = {}
    dA = dA_last
    for i in range(len(activations), 0, -1):
        dW, db, dA_prev = layer_backward_pass(dA,
                                              forward_pass_results['Z' + str(i)],
                                              forward_pass_results['A' + str(i-1)],
                                              parameters['W' + str(i)],
                                              activations[i-1])

        gradients['dW' + str(i)] = dW
        gradients['db' + str(i)] = db
        dA = dA_prev
    return gradients

def train_model(X, Y, layer_dimensions, layer_activations, initializations,
                learning_rate = 0.01, epochs = 10, batch_size = 64, lambd = 0,
                show_cost = False):

    m = X.shape[1]
    #Nombre de couches - hors celle des entrées
    l = len(layer_dimensions)-1

    # Création de tous les paramètres
    # A chaque étape, W a pour dimensions "nb neurones de la couche" x "nb entrées"
    # Et b est un vecteur, une valeur par neurone
    parameters = {}
    for i in range(1, l+1):
        parameters['W' + str(i)] = init_functions[initializations[i-1]](layer_dimensi
        parameters['b' + str(i)] = np.zeros((layer_dimensions[i], 1))

    costs = []
    # Apprentissage
    for e in range(epochs):
        for k in range(0, m, batch_size):
            x_batch = X[:, k:k+batch_size]
            y_batch = Y[:, k:k+batch_size]

            # Passe en avant
            forward_pass_results = model_forward_pass(x_batch, layer_activations, par

            # Calcul de la dérivée du coût par rapport au dernier A
            A_last = forward_pass_results['A' + str(l)]
            dA_last = -(np.divide(y_batch, A_last) - np.divide(1 - y_batch, 1 - A_last

            # Calcul des gradients - passe en arrière
            gradients = model_backward_pass(dA_last, parameters, forward_pass_results,

            # Descente de gradient
            for i in range(1, l+1):
                # Ajout de la régularisation L2
                parameters['W' + str(i)] -= learning_rate * (gradients['dW' + str(i)]
                parameters['b' + str(i)] -= learning_rate * gradients['db' + str(i)]

        # On peu de debug
        model_result = model_forward_pass(X, layer_activations, parameters['A' + str
        cost = np.sum(np.square(np.subtract(Y, model_result) * Y * np.log(1 - model_result
        regularization_cost = lambd/2/m * np.sum([np.sum(parameters['W'+str(i)]**2) fo
        cost += regularization_cost
        costs.append(cost)
        if show_cost :
            if e % (epochs // 20) == 0: print("Epoch #%i: %f" % (e, cost))

    return parameters, costs
```

Retour à la mise en situation

Un exemple de ce qu'il ne faut pas faire

On a vu que traiter l'overfit sur notre modèle de traitement des chiffres était une mauvaise idée. On va le confirmer.

C'est toujours le dataset de Yann Le Cun http://yann.lecun.com/exdb/mnist/ (images 28x28, 60.000 données d'entraînement et 10.000 données de validation).

In [3]:
```
def load(file):
    data = np.load(file)
    return data['x'], data['y']

x_train, y_train = load('data/609_train_data.npz')
x_test , y_test  = load('data/609_test_data.npz')

mus   = x_train.mean(axis = 0, keepdims = True)
sigmas = x_train.std (axis = 0, keepdims = True) + 1e-9

x_train_norm = (x_train-mus)/sigmas
x_test_norm  = (x_test -mus)/sigmas

y_train_mat = (y_train == np.arange(10)).astype(int)
```

Allez, c'est parti. On va tester un réseau avec, et sans régularisation L2.

In [4]:
```
def accuracy(x, y, params, layer_activations):
    results = np.argmax(model_forward_pass(x, layer_activations, params)['A'+str(len(
    return np.mean(results == y)

def test_reg(epochs, lambd = 0) :
    np.random.seed(0)
    activations = ['relu', 'relu', 'sigmoid']
    params, costs = train_model(x_train_norm.T, y_train_mat.T, [28*28, 50, 25, 10], ac
                                ['xavier', 'xavier', 'xavier'], lambd = lambd,
                                epochs = epochs, learning_rate = 0.005, show_cost = False
    print('Accuracy on training set : %f%%' % (100*accuracy(x_train_norm.T, y_train.T,
    print('Accuracy on test set : %f%%'     % (100*accuracy(x_test_norm.T , y_test.T ,
```

In [5]:
```
test_reg(epochs = 20)
```

```
Accuracy on training set : 96.906667%
Accuracy on test set : 96.040000%
```

```
<ipython-input-2-8fbb3e7d7b5f>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

In [6]:
```
test_reg(epochs = 20, lambd = 3)
```

```
Accuracy on training set : 92.288333%
Accuracy on test set : 92.280000%
```

```
<ipython-input-2-8fbb3e7d7b5f>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

Quel est le résultat de notre idée ? Il a essayé de moins coller aux données d'entraînement, et résultat on fait moins bien. Et sur le test aussi :(

Et si on persiste et qu'on continue sur cette voie, on peut toujours réduire le poids de la régularisation :

In [7]:
```
test_reg(epochs = 20, lambd = 1)
```

```
Accuracy on training set : 95.041667%
Accuracy on test set : 94.570000%
```

```
<ipython-input-2-8fbb3e7d7b5f>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

In [8]:
```
test_reg(epochs = 20, lambd = .1)
```

```
Accuracy on training set : 96.746667%
Accuracy on test set : 95.960000%
```

```
<ipython-input-2-8fbb3e7d7b5f>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

On ne fait au final que dégrader les performances en training, et avec le même impact sur le test. Un beau gachis.

On va maintenant revenir aux chatons. On faisait 100% au training et 74% au test, donc un vrai souci à pour le coup.

In [9]:
```
x_train_cat, y_train_cat = load('data/620_train_data.npz')
x_test_cat , y_test_cat  = load('data/620_test_data.npz')

mus   = x_train_cat.mean(axis = 1, keepdims = True)
sigmas = x_train_cat.std (axis = 1, keepdims = True) + 1e-9

x_train_cat_norm = (x_train_cat-mus)/sigmas
x_test_cat_norm  = (x_test_cat -mus)/sigmas

def accuracy_cat(x, y, params, layer_activations):
    results = model_forward_pass(x, layer_activations, params)['A'+str(len(layer_acti
    return np.mean(results == y)

def test_reg_cat(epochs, lambd = 0) :
    np.random.seed(0)
    activations = ['relu', 'relu', 'sigmoid']
    params, costs = train_model(x_train_cat_norm, y_train_cat, [64*64*3, 20, 7, 1], ac
                                ['he', 'he', 'he'], lambd = lambd, batch_size = 256,
                                epochs = epochs, learning_rate = 0.01, show_cost = False)
    print('Accuracy on training set : %f%%' % (100*accuracy_cat(x_train_cat_norm, y_tr
    print('Accuracy on test set : %f%%'     % (100*accuracy_cat(x_test_cat_norm , y_te
```

In [10]:
```
test_reg_cat(200)
```

```
Accuracy on training set : 99.521531%
Accuracy on test set : 70.000000%
```

In [11]:
```
test_reg_cat(200, lambd = 10)
```

```
Accuracy on training set : 99.043062%
Accuracy on test set : 72.000000%
```

In [12]:
```
test_reg_cat(200, lambd = 30)
```

```
Accuracy on training set : 99.043062%
Accuracy on test set : 76.000000%
```

In [13]:
```
test_reg_cat(200, lambd = 50)
```

```
Accuracy on training set : 99.043062%
Accuracy on test set : 78.000000%
```

In [14]:
```
test_reg_cat(200, lambd = 100)
```

```
Accuracy on training set : 97.607656%
Accuracy on test set : 76.000000%
```

In [15]:
```
test_reg_cat(200, lambd = 1000)
```

```
Accuracy on training set : 65.550239%
Accuracy on test set : 34.000000%
```

Là, pour le coup, c'est mieux. On n'arrive pas à réduire tout l'overfit à coup de régularisation L2, mais enfin on peut gagner jusqu'à 8 points en test pour le même réseau !

Par contre, à la fin, on voit aussi que trop, c'est trop :)

D'autres méthodes pour contrer l'overfit existent, comme le dropout, ou tout simplement avoir plus de données (y a que 200 chats en training...), un réseau moins large ou moins profond, ...