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

Xavier, He, Bengio

Mise en situation

J'ai un réseau mutlticouches : ReLU, ReLU, ReLU, ..., et Sigmoïde à la fin.

Je suis incapable de faire tourner une descente de gradient. Ça plante avant même la première étape, et pourtant mon code est bon !

L'explosion / la disparition du signal

On a mentionné très brièvement le principe déjà, mais voilà le détail.

W trop grands

On va considérer pour l'exemple un réseau de taille "Input 64x64x3" / "ReLU 20" / "ReLU 10" / "ReLU 5" / "Sigmoid 1" .

Supposons que je prenne mes valeurs initiales de W entre 0 et 1, uniformément : la valeur moyenne est de 0.5 statistiquement. Prenons aussi une donnée d'exemple avec toutes les mesures égales à 1.

- Sur le passage de la première couche
  - Z va sommer 64x64x3 = 12288 termes. Total moyen : 6144 en moyenne sur chacun des 20 neurones.
  - ReLU : on à 6144 en sortie
- Couche suivante :
  - Calcul de Z : 6144 x 20 termes qui valent 0.5 en moyenne, total : 61440
  - ReLU : 61440
- Couche suivante (j'abrège) : 307200 en moyenne
- Sigmoide finale : s(768000) ~= 1

Le résultat est $1-\epsilon$ avec $\epsilon\ = 0.000000000...0000....000....0006$, enfin bref il va falloir plus de 330.000 zéros avant d'avoir un chiffre significatif dans ϵ. **Autant dire que la sigmoide vaut 1, point barre**, en tout cas Python le définira comme tel.

Arrive le calcul du gradient, qu'on commence sur la dernière couche : $dA = -\frac{1}{m}(\frac{Y}{A} - \frac{1-Y}{1-A})$: la deuxième fraction avec son dénominateur 1-A va faire une division par 0 (ou presque si on est puriste, mais pour Python ça ne fera pas une grosse différence...)

W trop petits

Mauvaise solution : je divise tous mes poids de départ par 500 millions et je suis tranquille. Valeur moyenne d'un poids : 1 milliardième.

Le résultat, c'est que le signal risque de disparaitre : Prenons "Input 2" / "ReLU 2" / "ReLU 2" / "ReLU 2" / "ReLU 2" / "Sigmoid 1" :

- en sortie de la première couche : Z = A = 2e-9 en moyenne
- en sortie de la seconde : Z = A = 4e-18
- puis Z = A = 8e-27
- puis Z = A = 16e-36
- en sortie de la dernière : 0.5 à un micro pouillème près - parce que 0.000000........016 ou rien c'est pareil...

Il faut donc trouver une méthode smart d'initialisation

Initialion des poids

Distribution

Déjà, au lieu de prendre une distribution uniforme, on va prendre une distribution normale. Ça permet d'être plus exact quand on fait des hypothèses du genre "la moyenne devrait être autour de " etc...

Pour rappel, une distribution est définie par sa moyenne μ et sa variance σ.

Initialisation de Xavier

Très utile dans les tanh : il est nécessaire d'avoir des valeurs pas trop éloignées de 0 pour que la tanh renvoie autre chose que 1 ou -1 (au pouillème près).

Les n données d'entrée étant elles-mêmes normalisées, il a été montré que prendre une variance $\frac{1}{n}$ permettait d'alimenter tanh avec des valeurs correctes. Il faut donc, pour obtenir cette variance, diviser les valeurs aléatoire obtenues par l'écart-type, soit $\frac{1}{\sqrt{n}}$

Initialisation de He

Une variante, qui marche mieux pour les fonctions ReLU : appliquer un facteur 2 à la variance de Xavier, et donc un facteur $\sqrt{2}$ à l'écart-type : $\sqrt{\frac{2}{n}}$

Initialisation de Bengio

De Xavier Glorot et Bengio, en fait; mais comme la première méthode elle-même est déjà dénommée d'après Xavier Glorot... on s'y perd :)

Bref, là, l'idée est de prendre cette fois $\frac{1}{\sqrt{n_1+n_2}}$, avec n_1 la taille de la couche précédente (taille des entrées) et n_2 la couche elle-même (taille de la sortie)

Autres

Vu que ces découvertes sont assez récentes, il risque fort d'y en avoir d'autres dans un avenir pas si lointain :)

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_

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):
    s = sigmoid(x)
    return s * (1 - s)

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,
                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_dimensio
        parameters['b' + str(i)] = np.zeros((layer_dimensions[i], 1))

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

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

            # 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):
                parameters['W' + str(i)] -= learning_rate * gradients['dW' + str(i)]
                parameters['b' + str(i)] -= learning_rate * gradients['db' + str(i)]

        # Un peu de debug
        model_result = model_forward_pass(X, layer_activations, parameters)['A' + str
        cost = np.squeeze(-np.sum(np.log(model_result) * Y + np.log(1 - model_result)
        costs.append(cost)
        if show_cost : print('Epoch #%i: %s' % (e+1, cost))
    return parameters, costs
```

Retour à la mise en situation

Chargement des données

On continue avec le dataset de Yann Le Cun http://yann.lecun.com/exdb/mnist/ (images 28x28, 60.000 données d'entrainement et 10.000 données de validation), et on va regarder les impacts sur un réseau

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

x_train, y_train = load('data/d09_train_data.npz')
x_test , y_test  = load('data/d09_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 reprendre notre réseau pas trop mal (cf jour 19) : "relu 50 / relu 25 / sigmoide 10"

In [4]:
```
activations = ['relu', 'relu', 'sigmoid']

def test_initializations(initializations, epochs) :
    np.random.seed(0)
    return train_model(x_train_norm.T, y_train_mat.T, [28*28, 50, 25, 10], activation
                       initializations,
                       epochs = epochs, learning_rate = 0.005, show_cost = True)
```

In [5]:
```
params, costs = test_initializations(['uniform', 'uniform', 'uniform'], epochs = 1)
```

```
<ipython-input-2-00fa822fa5c7>:108: RuntimeWarning: divide by zero encountered in true
_divide
  dA_last = -(np.divide(y_batch, A_last) - np.divide(1 - y_batch, 1 - A_last))/x_batc
h.shape[1]
<ipython-input-2-00fa822fa5c7>:108: RuntimeWarning: invalid value encountered in true_
divide
  dA_last = -(np.divide(y_batch, A_last) - np.divide(1 - y_batch, 1 - A_last))/x_batc
h.shape[1]
<ipython-input-2-00fa822fa5c7>:59: RuntimeWarning: invalid value encountered in multip
ly
  dZ = dA * act_derivates[activation](Z)
Epoch #1: nan
```

Boum. Même pas un début de quelque chose, le système est déjà mort. Passer par des distributions uniformes, c'est bof. On va tester la division par une constante (100)

In [6]:
```
params, costs = test_initializations(['uniform_100', 'uniform_100', 'uniform_100'], ep
```

```
Epoch #1: 3.4451083524077366
Epoch #2: 3.2095969256610655
Epoch #3: 2.9372929160907137
Epoch #4: 2.722872978659204
Epoch #5: 2.2754168865291873
Epoch #6: 1.8077371861216853
Epoch #7: 1.4202047548039582
Epoch #8: 1.2050923686359423
Epoch #9: 0.9899931903889271
Epoch #10: 0.7827429621616919
Epoch #11: 0.6483106040722668
Epoch #12: 0.5664563399518844
Epoch #13: 0.5102771884965723
Epoch #14: 0.4679848803323316
Epoch #15: 0.4345764114964007
Epoch #16: 0.4070456281168636
Epoch #17: 0.3836346298939539
Epoch #18: 0.36350435259368297
Epoch #19: 0.34586114861486666
Epoch #20: 0.3302383516982213
```

In [7]:
```
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)

print('Accuracy on training set : %f%%' % (100*accuracy(x_train_norm.T, y_train.T, pa
print('Accuracy on test set : %f%%'     % (100*accuracy(x_test_norm.T , y_test.T , pa
```

```
Accuracy on training set : 95.281667%
Accuracy on test set : 94.610000%
<ipython-input-2-00fa822fa5c7>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

Le résultat est plutôt très bon, et avec 20 itérations seulement.

On teste maintenant une simple distribution gaussienne (sans facteur multiplicateur).

In [8]:
```
params, costs = test_initializations(['gaussian', 'gaussian', 'gaussian'], epochs = 1
```

```
<ipython-input-2-00fa822fa5c7>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
<ipython-input-2-00fa822fa5c7>:108: RuntimeWarning: divide by zero encountered in true
_divide
  dA_last = -(np.divide(y_batch, A_last) - np.divide(1 - y_batch, 1 - A_last))/x_batc
h.shape[1]
<ipython-input-2-00fa822fa5c7>:108: RuntimeWarning: invalid value encountered in true_
divide
  dA_last = -(np.divide(y_batch, A_last) - np.divide(1 - y_batch, 1 - A_last))/x_batc
h.shape[1]
<ipython-input-2-00fa822fa5c7>:59: RuntimeWarning: invalid value encountered in multip
ly
  dZ = dA * act_derivates[activation](Z)
Epoch #1: nan
```

Pareil que la distribution uniforme, ça ne tient pas le premier round !

In [9]:
```
params, costs = test_initializations(['xavier', 'xavier', 'xavier'], epochs = 20)
print('Accuracy on training set : %f%%' % (100*accuracy(x_train_norm.T, y_train.T, pa
print('Accuracy on test set : %f%%'     % (100*accuracy(x_test_norm.T , y_test.T , pa
```

```
Epoch #1: 1.0125611633216307
Epoch #2: 0.6442987536397736
Epoch #3: 0.5307315466860846
Epoch #4: 0.469224625860632
Epoch #5: 0.4274914057945546
Epoch #6: 0.39618900427008896
Epoch #7: 0.37098109568591586
Epoch #8: 0.35007864626282337
Epoch #9: 0.3322431526405294
Epoch #10: 0.31664505587135205
Epoch #11: 0.3028466467498225
Epoch #12: 0.2904549703974487
Epoch #13: 0.2791373956663525
Epoch #14: 0.2687646284683848
Epoch #15: 0.2593010920991946
Epoch #16: 0.25051651454632606
Epoch #17: 0.2423501186660231
Epoch #18: 0.2347128040883857
Epoch #19: 0.22753375477933183
Epoch #20: 0.22078085498835542
Accuracy on training set : 96.906667%
Accuracy on test set : 96.040000%
<ipython-input-2-00fa822fa5c7>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

Là c'est très intéressant : on voit que dès la première étape le cout est 3 fois moindre que tout à l'heure, on a une convergence beaucoup plus rapide !

In [10]:
```
params, costs = test_initializations(['he', 'he', 'he'], epochs = 20)
print('Accuracy on training set : %f%%' % (100*accuracy(x_train_norm.T, y_train.T, pa
print('Accuracy on test set : %f%%'     % (100*accuracy(x_test_norm.T , y_test.T , pa
```

```
Epoch #1: 0.8579472765944866
Epoch #2: 0.6065799506694717
Epoch #3: 0.5118657046569804
Epoch #4: 0.4561725435260324
Epoch #5: 0.41769774044723335
Epoch #6: 0.3883302562777395
Epoch #7: 0.36475675226187526
Epoch #8: 0.34520759268055756
Epoch #9: 0.3286283269816935
Epoch #10: 0.31406060454078205
Epoch #11: 0.30110103335478156
Epoch #12: 0.2894717821576476
Epoch #13: 0.27892852008999425
Epoch #14: 0.26919729023451905
Epoch #15: 0.26018325633595996
Epoch #16: 0.2518317360532692
Epoch #17: 0.2440085357983924
Epoch #18: 0.236706242807445
Epoch #19: 0.2299390923875491
Epoch #20: 0.22347550008250824
Accuracy on training set : 96.903333%
Accuracy on test set : 95.790000%
<ipython-input-2-00fa822fa5c7>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

In [11]:
```
params, costs = test_initializations(['bengio', 'bengio', 'bengio'], epochs = 20)
print('Accuracy on training set : %f%%' % (100*accuracy(x_train_norm.T, y_train.T, pa
print('Accuracy on test set : %f%%'     % (100*accuracy(x_test_norm.T , y_test.T , pa
```

```
Epoch #1: 1.2023023886461857
Epoch #2: 0.702118236400899
Epoch #3: 0.5636476647508493
Epoch #4: 0.4943163394964271
Epoch #5: 0.44897452009644767
Epoch #6: 0.4156369027593031
Epoch #7: 0.3890658982109316
Epoch #8: 0.3669035000293687
Epoch #9: 0.34813315060624606
Epoch #10: 0.33179756328079835
Epoch #11: 0.3172723735420891
Epoch #12: 0.30425873103889717
Epoch #13: 0.29243153115213366
Epoch #14: 0.28171989983741247
Epoch #15: 0.271788224535598
Epoch #16: 0.2626436164822237
Epoch #17: 0.25410711693936927
Epoch #18: 0.24605809468439763
Epoch #19: 0.23850388108851514
Epoch #20: 0.23135582360542828
Accuracy on training set : 96.770000%
Accuracy on test set : 95.960000%
<ipython-input-2-00fa822fa5c7>:17: RuntimeWarning: overflow encountered in exp
  def sigmoid(x) : return 1 / (1 + np.exp(-x))
```

Voilà. On constate que suivant les méthodes d'initialisation, on converge plus ou moins vite, les premières étapes sont plus ou moins optiomales, etc...

Pour savoir laquelle est la meilleure... ça dépendra beaucoup du sujet. Faut tester :) Là c'est la méthode Xavier qui passe bien, mais en pratique su run réseau de taille "utile" on trouvera en général que He ou Bengio fonctionne mieux.

Mais en tous les cas, on a quand même plus de 95% de succès à chaque fois avec 20 itérations uniquement !