Ce TP a pour but de vous familiariser avec l'utilisation d'un réseau de neurones. Nous allons ici implémenter un réseau de neurones qui dont l'objectif sera de reconnaître des lettres grecques. Ce TP se divise en trois parties : La première où l'on implémentera un réseau de neurones simple (zéro couches cachées) la seconde où l'on ajoutera une couche cachée et la dernière où vous devrez optimiser le réseau. Vous trouverez sur les pages suivantes une explication de ce qu'est un réseau de neurones et la régression logistique.
`www.math.univ-toulouse.fr/~gadat/Ens/M2SID/08-m2-regression_logistique.pdf`
`https://fr.wikiversity.org/wiki/R%C3%A9seaux_de_neurones`,

Pour faire simple la régression logistique est un réseau de neurones sans couches cachées dont la fonction d'activation est la fonction softmax. Chaque neurone de sortie cherche à prédire la probabilité d'appartenance à la classe qu'il représente. C'est le classifieur le plus simple qui a une structure de réseau de neurones. Le calcul de l'erreur du résultat ne se fait alors pas directement sur les classifications des entrées, mais utilise le concept de vraisemblance emprunté du domaine des probabilités.

Pour ce TP Nous allons utiliser Python qui offre des librairies de calcul adaptées à l'encodage d'un réseau de neurones. Il faudra pour ce TP installer la librairie Theano de Python. Vous trouverez toute les informations nécessaires sur le lien suivant : `http://www.deeplearning.net/software/theano/install.html`. Pour vous familiariser avec cette librairie de calcul formel je vous conseille de consulter le tutoriel au lien suivant `http://www.deeplearning.net/software/theano/tutorial/index.html#tutorial`.

1 Les données

Vous allez entrainer les algorithmes de ce TP sur les données présentes sur le moodle. Vous y trouverez deux fichiers : un fichier dataX.npy et un fichier dataY.npy dataX contient une matrice de class numpy (`https://docs.scipy.org/doc/numpy/`) de taille N*1024. Chaque ligne de cette matrice représente une image 32*32 pixels. Les images représentent des lettres de l'alphabet grec dans une position quelconque. Le classifieur que vous aller mettre en place cherchera à déterminer de quelle lettre il s'agit. DataY fourni le label de chaque image (lettre représentée). Vous pouvez chercher à afficher les lettres si vous le souhaitez, en utilisant :

```
import matplotlib
import matplotlib.pyplot as plt

cmap = matplotlib.cm.Greys
plt.imshow(X[k].reshape((32,32)),cmap=cmap)
```

Où X est un tableau numpy contenant toutes les données

2 Implémentation de la régression logistique

Nous allons commencer par implémenter un classificateur par réseau de neurones sans couches cachées. Ce dispositif d'apprentissage est équivalent à une régression logistique. Des classes python vous sont fournies pour vous simplifier la tâche.

2.1 Complétion de la classe LogisticRegression

Complétez le champ $self.p\_y\_given\_x$ de la classe LogisticRegression. Utilisez pour cela les fonctions de Theano $theano.tensor.nnet.softmax$ et $theano.tensor.dot$ qui sont respectivement les fonctions softmax et produit matriciel de Theano (sur les tenseurs)

Pour rappel :

$$P(Y_i|X) = \frac{e^{(W.X+b)_i}}{\sum_j e^{(W.X+b)_j}}$$

Complétez ensuite le champ $self.y\_pred$ qui contient la valeur des prédiction (argmax des probabilités). Theano a aussi une fonction $argmax$. On prédit la classe avec la plus forte probabilité.

Complétez la fonction $loss(self, y)$ qui calcule la log vraisemblance des prédictions. Vous aurez besoin pour cela des fonctions mean, log et arrange de Theano. Pour rappel

$$loss(Y = (Y_i)) = -\sum_i ln(P(Y_i|X_i))$$

où $Y = (Y_i)$ représente les résultats connus, et $X = (X_i)$ les entrées correspondantes

Complétez la fonction $errors(self, y)$ qui calcule le taux d'erreurs. Vous pouvez utiliser les fonctions mean et neq de theano. Utilisez une erreur quadratique moyenne

$$err = \frac{1}{n}\sum (y_{pred} - y_i)^2$$

2.2 Complétion de la fonction Load_data

Cette fonction charge les données d'entraînement en mémoire et retourne un résultat de la forme : $(train\_set, valid\_set, test\_set)$ où train_set, valid_set et test_set sont trois couples contenant une matrice X d'entrées et un vecteur Y de valeurs.

Cette classe contient déjà une fonction $shared\_dataset(dataxy)$ qui permet de créer des données statiques, pour accélérer l'entraînement du modèle (surtout sur CPU). elle transforme les tableaux numpy en tenseurs partagés theano.

2.3 Implémentation de la régression

C'est la fonction clé du TD, celle qui implémente la régression logistique et donc l'apprentissage.
Il faut commencer par charger les données en appelant la fonction *load_data* et les assigner à des variables.

2.3.1 Définition du modèle

Définir les variables abstraites theano du problème :

— le scalaire $index = T.lscalar()$; un entier indiquant le numéro de l'échantillon en cours de traitement

— la matrice $x = T.matrix('x')$ qui contiendra les données d'entrée

— le vecteur $y = T.vector('y')$ qui contiendra les classes des données

Ces objets sont abstraits et ne contiennent rien pour l'instant. Ce sont des variables comme x en mathématiques. Ils seront évalués à l'exécution du code
Construisez maintenant un objet de type logistic_regression, qui représentera l'opération de calcul de la régression, la calculer pour l'instant sur la matrice x abstraite et le coût $cost = log\_reg.loss(y)$, pris sur le vecteur abstrait y.

Construire aussi les fonctions *train_model*, *test_model* et *validate_model* se basant sur les objets abstraits x, y et *index*
Ces trois fonctions sont des fonctions Theano (voir tutoriel) qui prennent en argument un index (index de l'échantillon courant) et qui calculent, pour le train, la perte et met à jour le modèle, pour le validate et le test, l'erreur commise sur l'échantillon d'apprentissage.
Par exemple la fonction test_model prendra comme argument :

— input=[index]

— outputs=log_reg.errors(y)

— givens = {
x= test_set_x[index*taille de l'échantillon :(index+1)*taille de l'échantillon],
y= test_set_y[index*taille de l'échantillon :(index+1)*taille de l'échantillon]
}

La fonction validate_model est très similaire à celle de test. Pour la fonction train_model il faut d'abord définir le mise à jour des paramètres. Il faut compléter la variable updates qui a la forme suivante : [(param1,nv_param1),...] et qui calcule la mise à jour des paramètres du modèle à partir de la perte.
Déterminez la fonction qui étant donnée l'erreur calcule la mise à jour des paramètres du problème. Vous pouvez utiliser dans cette question la fonction *grad* de theano qui calcule le gradient par rapport à un paramètre d'une la fonction theano à partir des données.

Complétez maintenant la fonction train_model :

```
train_model = theano.function(
        inputs=[index],
```

```
        outputs=cost,
        updates=updates,
        givens={
            x: train_set_x[index * taille de l'échantillon:
             (index + 1) * taille de l'échantillon],
            y: train_set_y[index * taille de l'échantillon:
             (index + 1) * taille de l'échantillon]
        }
    )
```

2.3.2 Entraînement

Il reste maintenant à entraîner le modèle. Les fonctions définies auparavant vont grandement simplifier notre travail :
Il suffit pour chaque échantillon d'appeler la fonction *train_model* avec le bon argument. Puis régulièrement, par exemple à chaque epoch (parcours de toutes les données une fois), il faut calculer l'erreur commise sur l'ensemble de validation. On arrête l'entraînement au bout d'un nombre prédéfinis d'epochs (30 par exemple) On teste alors le modèle obtenu sur l'ensemble de test.
Les implémentations proposées divisent le test et le train en échantillons, mais cela n'est pas obligatoire (vous pouvez tester sur tout l'ensemble en même temps).

2.3.3 Terminaison

De nombreux critères de terminaison existent, je vous invite à explorer différentes pistes. Une idée simple est de tester si l'erreur est stabilisée et arrêter dès qu'elle ne varie que très peu.
Testez au moins deux critères de terminaisons et discutez leur avantages et leur défauts. Les critères peuvent être inefficaces, s'il le sont expliquez pourquoi et comment les améliorer.

3 Réseau de neurones avec une couche cachée

Dans cette partie on va ajouter à notre réseau de neurones une couche cachée, ce qui nous fait passer d'une régression logistique à un réseau de neurones classique.

3.1 Complétion de la classe Hiddenlayer

Complétez la classe HiddenLayer, qui contient uniquement un constructeur qui construit les deux paramètres de la couche et les stocke dans self.W et self.b et calcule la sortie self.output (choisir une fonction d'activation softmax, tanh, ...).

3.2 Complétion de la classe NN

La classe NN va représenter notre réseau de neurones, elle va donc encapsuler l'entrée, la couche cachée et la couche de sortie (régression logistique) Complétez les champs :

— input : la matrices des données en entrée
— hiddenlayer : un élément du type hidden layer
— logreg : un élément du type logreg
— loss : la perte (négative log vraisemblance) du résultat
— params = [W_hidden, b_hidden, W_logreg, b_logreg] : les paramètres du réseau

3.3 Entraînement du réseau de neurones

Modifiez la fonction d'apprentissage pour entraîner un réseau de neurones :

— Remplacez la régression logistique par le réseau neurones et corrigez toutes les dépendances
— Modifiez la fonction de calcul de rétro propagation (argument updates de la fonction train_model) de l'erreur pour mettre à jour la couche cachée.

4 Optimisation

Testez différents paramètres sur votre modèle et trouvez ceux qui aboutissent au meilleur modèle. Vous pouvez pour cela jouer sur :

— la taille des couches
— le nombre de couches cachées
— le taux d'apprentissage
— la fonction d'activation

5 Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichiers .zip par groupe, nommés TD1_nom_prenom_matricule.zip. Vous devez remettre un fichier pdf contenant une explication de vos implémentations ainsi que tous les commentaires que vous jugerez utiles et le code Prolog. La clarté et la concision du rapport seront pris en compte. Tout devra être remis avant le **18 avril à 23h55**. Tout travail en retard sera pénalisé d'une valeur de 10% par jour de retard.

Barème :

Section 2 :	50pts
Section 3 :	30pts
Section 4 :	10pts
rapport :	10pts

*Ce TP est une repise du TP2 de l'hiver 2016 complété par un projet imaginé par Pierre Hulot