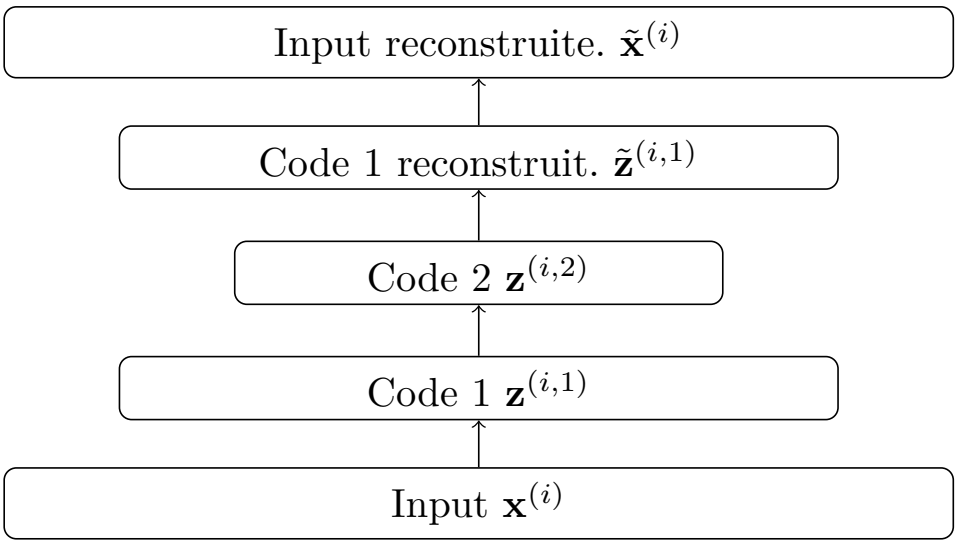
DatFus - TP n°3

Dans ce 3ème TP, on s'intéresse à un modèle d'apprentissage de fusion populaire : le **stacking**. Néanmoins, nous allons utiliser cette approche ici dans un contexte un peu éloigné de la fusion à savoir celui des **stacked auto-encoders**.

Un **auto-encoder** est un type particulier de réseaux de neurones dont le but est de reconstruire les inputs $\mathbf{x}^{(i)}$ en passant par plusieurs représentations intermédiaires (ou codes) correspondant aux sorties des couches. La fonction objectif minimisée par descente de gradient lors de l'apprentissage par l'algorithme de rétro-propagation est l'erreur quadratique moyenne. Si $\tilde{\mathbf{x}}^{(i)}$ est l'input reconstruite par le réseau, la fonction objectif est

$$\sum_{i=1}^{n_{\text{train}}} \left\| \tilde{\mathbf{x}}^{(i)} - \mathbf{x}^{(i)} \right\|. \tag{1}$$

Un **auto-encoder** est un réseau symétrique et généralement, les poids des couches symétriques sont les mêmes (à une transposée près). Voici, l'architecture sur laquelle nous allons travailler aujourd'hui :



Dans ce réseau, le passage d'un niveau de représentation à l'autre se fait toujours à l'aide d'une opération matricielle faisant intervenir les paramètres d'une couche neuronale. Par exemple, pour passer de $\mathbf{x}^{(i)}$ à $\mathbf{z}^{(i,1)}$, on utilise les poids $\mathbf{W}^{(1)}$ et les *bias* $\mathbf{b}^{(1)}$ correspondant à la première couche neuronale selon :

$$\mathbf{z}^{(i,1)} = f_{\text{act}}\left(\mathbf{W}^{(1)} \cdot \mathbf{x}^{(i)} + \mathbf{b}^{(1)}\right), \tag{2}$$

où f_{act} est une fonction d'activation. Le nombre de colonnes de la matrice $\mathbf{W}^{(1)}$ est égal à la taille du vecteur $\mathbf{x}^{(i)}$. Le nombre de lignes de la matrice $\mathbf{W}^{(1)}$ ainsi que la taille du vecteur $\mathbf{b}^{(1)}$ sont égaux au nombre d'unités neuronales de la couche (un hyperparamètre à choisir).

Notre but est d'apprendre des matrices de poids et des vecteurs de *bias* pour les 4 couches reliant nos 5 représentations, de sorte que la reconstruction des inputs soit la meilleure possible. Nous allons légèrement simplifier la problématique en liant les matrices de poids des couches symétriques selon :

$$\mathbf{W}^{(4)} = {\mathbf{W}^{(1)}}^T, \tag{3}$$

$$\mathbf{W}^{(3)} = {\mathbf{W}^{(2)}}^T. \tag{4}$$

On parle alors de *tied weights*. Ce choix n'est pas absolument nécessaire mais il permet de réduire le nombre de paramètres à apprendre tout en gardant un modèle flexible.

Mais où est la fusion dans tout ça ? Premièrement, une couche d'un réseau de neurones peut tout à fait se voir comme un comité d'unités neuronales que l'on fusionne. Un réseau est aussi à ce titre une fusion hiérarchique. Habituellement, tous les paramètres d'un réseau sont appris en même temps grâce à l'algorithme de rétro-propagation. Pour un réseau de neurones, **stacking** propose d'entraîner des fusions couche par couche.

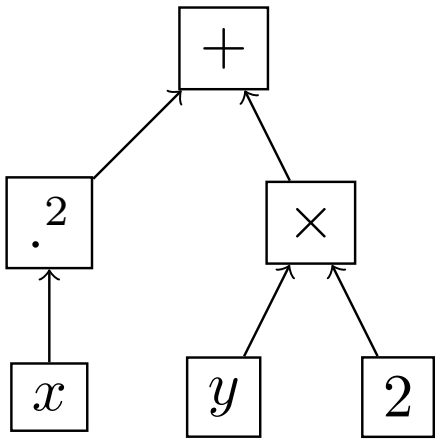
Pourquoi est-ce intéressant ? En réalité, tout cela sert un seul but : obtenir une très bonne initialisation des poids des couches basses d'un réseau profond. Le réseau complet se termine par un *multi-layer perceptron* (MLP). Nous verrons plus loin comment un MLP vient se greffer dans notre histoire d'auto-encoder. Quand l'auto-encoder est très profond, il devient coûteux de l'entraîner juste pour une histoire d'initialisation. C'est là qu'intervient le **stacking**, qui coûte r entraînements d'auto-encoder à une couche au lieu d'un entraînement d'un auto-encoder à r couches (plus cher). Il permet aussi de contourner la difficulté liée au gradient evanescent.

Exercice 1 : prise en main et comparaison à l'ACP

L'intérêt principal d'un auto-encoder est qu'il permet de calculer des codes dont la taille est plus petite que celle de l'input. C'est donc une méthode non-supervisée de réduction de dimension. Quand un auto-encoder a une seule couche, son modèle est d'ailleurs très proche de l'ACP qui est aussi construite à partir de minimisation de l'erreur quadratique moyenne.

Dans cet exercice, nous allons illustrer la capacité d'un auto-encoder à opérer une réduction de dimension sur un dataset jouet. Pour programmer un auto-encoder, nous allons utiliser `TensorFlow`. Cette librairie est très commode pour construire des réseaux de neurones (potentiellement profonds) tout en répartissant la charge de calcul sur différents CPUs ou GPUs.

`TensorFlow` est construite sur une logique de graphe de calcul. Voici un exemple d'un tel graphe :



Ce graphe correspondrait au calcul de la fonction

$$f(x, y) = x^2 + 2y.$$

Les variables x et y seront représentées par des instances de la classe `tf.placeholder`. On pourrait alors programmer f en une ligne de code python mais aucun de ces éléments ne prendrait vie avant de lancer une `session` de `TensorFlow`. En effet, on commence par décrire symboliquement le graphe de calcul et on effectue le calcul à proprement parler qu'à l'intérieur d'une `session`. Voici un code mettant en œuvre ce graphe sur trois couples (x, y) :

```
import tensorflow as tf
x = tf.placeholder(tf.float32,shape=[])
y = tf.placeholder(tf.float32,shape=[])
f = x*x + 2*y
init = tf.global_variables_initializer()
some_x = [1,2,3]
some_y = [2,3,4]
with tf.Session() as sess:
      init.run()
      for i in range(len(some_x)):
            print(sess.run(f,feed_dict={x: some_x[i],y: some_y[i]}))
```

La classe `tf.placeholder` définit une variable qui sera remplacée plus tard par un flot de données. Ici, on spécifie que ces données seront de type `tf.float32` et qu'il s'agit de scalaires (`shape=[]`). Plus loin, dans la session, lors de l'appel à la méthode `run`, on explique que `x` et `y` doivent être substituées à `some_x[i]` et `some_y[i]`.

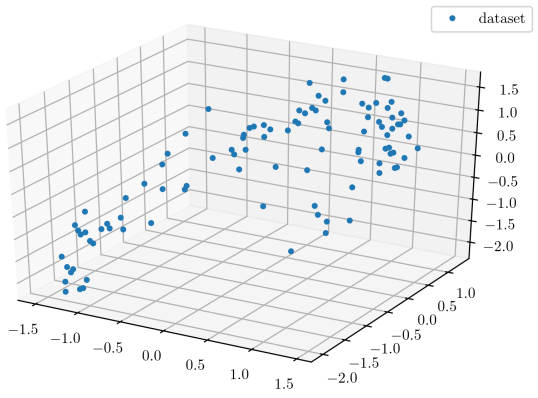
L'objet `init` sert à initialiser conjointement toutes les variables de `TensorFlow`. On peut initialiser individuellement chaque variable si besoin, mais une variable non initialisée ne peut être utilisée dans un calcul lors d'une `session`.

Pour mettre en œuvre les auto-encoders, nous aurons également besoin d'instance de la classe `tf.Variable` afin de représenter les matrices $\mathbf{W}^{(j)}$ et les vecteurs $\mathbf{b}^{(j)}$.

Questions :

1. En vous documentant sur `tf.gradients`, évaluez le gradient de f en les mêmes trois couples de points que dans le code précédent.

Nous allons à présent voir un premier auto-encoder rudimentaire qui réduit un dataset de points en 3D à une représentation en 2D.



Pour la suite de l'exercice, vous devez partir du code contenu dans le fichier `autoenc_etu.py`. Dans ce fichier, vous verrez que nous faisons appel à de nombreux outils de `TensorFlow` que nous n'avons pas le temps de présenter en détail :

— une initialisation spécifique pour les matrices $\mathbf{W}^{(j)}$ à l'aide de `tf.contrib.layers.variance_scaling_initializer`,
— l'algorithme d'optimisation ADAM à l'aide de `tf.train.AdamOptimizer`,
— un opérateur d'apprentissage obtenu en appelant la méthode `minimize` de l'instance de la classe `tf.train.AdamOptimizer`.

Questions :

2. Récupérez la représentation intermédiaire générée par l'auto-encoder pour tout le dataset et affichez le résultat avec un plot 2D.

3. Quelle fonction d'activation est utilisée ? Documentez vous sur ses spécificités.

4. Tracez l'évolution de l'erreur quadratique moyenne au fil des itérations.

5. Expliquez pourquoi d'un point de vue théorique, cette méthode n'est pas formellement équivalente à une ACP.

Exercice 2 : un auto-encoder multi-couches pour `MNIST`

Dans cet exercice, nous allons à présent programmer un auto-encoder à 3 couches (comme dans la figure en page 1) et l'entraîner sur le dataset `MNIST` qui contient des images de chiffres manuscrits. Les images sont en niveaux de gris et de taille 28×28. Ce dataset contient 65000 exemples d'apprentissage répartis en 55000 pour le train et en 10000 pour le test. Ici, on pourra omettre la *cross-validation* du fait de la taille importante du dataset qui garantit une évaluation des performances fiable. Le découpage train/test est déjà pré-programmé par `TensorFlow`.

Nous allons à présent partir du fichier `autoenc2_etu.py`. De nouveaux éléments font leur apparition dans ce code. Nous allons rajouter une pénalité L_2 à notre fonction objectif afin de régulariser le problème. On instancie la classe `tf.contrib.layers.l2_regularizer` puis on ajoute cet objet à la fonction objectif `J`. La régularisation s'applique aux matrices $\mathbf{W}^{(1)}$ et $\mathbf{W}^{(2)}$.

1. Le code du fichier `autoenc2_etu.py` génère une erreur quand vous l'exécutez au départ. Cela est du au fait que le graphe de calcul est incomplet. En effet, en vous inspirant du code de l'exercice précédent et de celui de la première couche, vous devez rajouter les couches 2, 3 ainsi que celle de sortie correspondant aux inputs reconstruites.

2. Affichez à l'aide d'un `print` l'erreur quadratique moyenne au fil des epochs.

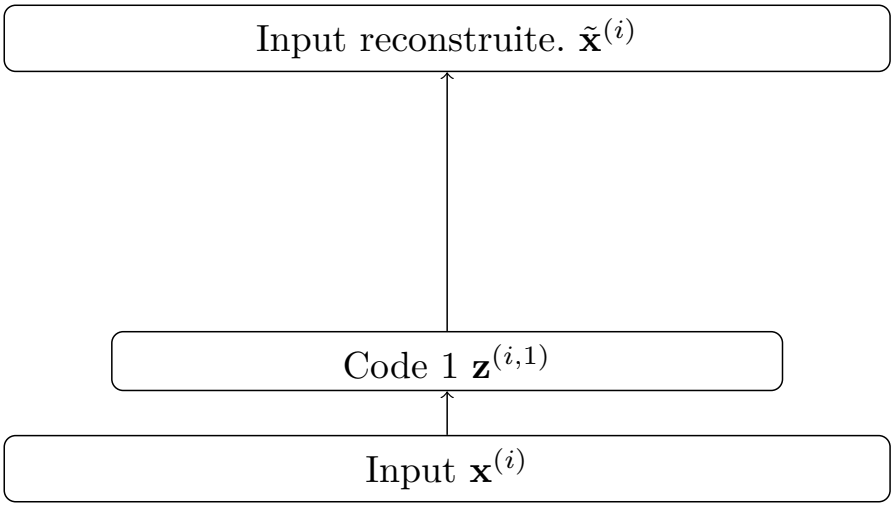
3. Obtenez les images reconstruites pour tout le *test set* et affichez un couple d'image avant/après reconstruction. Résultat attendu :

Exercice 3 : stacked auto-encoder et pre-training pour `MNIST`

Cet exercice suit directement le précédent. Nous entrons à présent dans le vif du sujet, à savoir le **stacking**. Nous allons entraîner le même réseau qu'à l'exercice 2 mais couche par couche. Une fois nos poids déterminés, nous rajouterons une couche de type *softmax regression* pour obtenir un MLP habituel. On lance ensuite l'algorithme de retro-propagation sur ce réseau dont une partie des poids sont intelligemment initialisés.

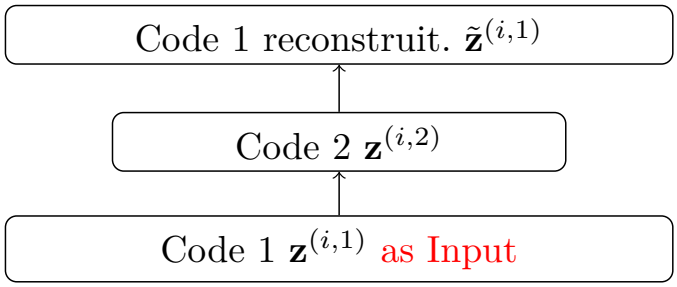
Nous partirons d'une trame de départ donnée par le fichier `autoenc3_etu.py`. Dans ce fichier, des lignes de code à compléter sont marquées par la présence du commentaire `#MISSING CODE`. Certains blocs de code sont à décommenter au fur et à mesure des questions.

1. Phase 1 : mettez en place le graphe correspondant à la figure suivante :



Le code pour la couche d'input ainsi que celui de la fonction objectif sont donnés. Il manque donc la première couche de codage et celle d'output.
Le code pour l'apprentissage dans la `session` est également donné et peut être lancé dès que le graphe est complété.

2. Générez les représentations intermédiaires $\mathbf{z}^{(i,1)}$ à l'aide la couche apprise pour tout le *train set*. Elles jouerons le rôle d'input pour entraîner la couche suivante. Vous les conserverez dans un `numpy array` appelé `Z_train`. Le code nécessaire à cette opération doit être inséré dans la `session`, après la boucle d'apprentissage de la phase 1.
3. Phase 2 : décommentez le bloc de code prévu pour le graphe de la phase 2 et correspondant à la figure suivante :



Le code pour la couche d'input (ici les $\mathbf{z}^{(i,1)}$) ainsi que celui de la fonction objectif sont donnés. Il manque donc la deuxième couche de codage et celle d'output (ici les $\tilde{\mathbf{z}}^{(i,1)}$).

4. Décommentez, la partie du code correspondant à la phase 2 dans la `session`. Il y a une seule ligne de code à compléter. Vous devez prélever un mini-batch dans `Z_train`.
5. Rassemblement ! Décommentez le bloc de code prévu pour le graphe complet. Dans ce graphe, on peut directement ré-utiliser le `tf.placeholder` correspondant aux inputs $\mathbf{x}^{(i)}$ ainsi que la première couche d'encodage. En revanche, vous devez recréer de nouvelles couches pour les 2 suivantes et celle d'output. Par contre, vous donnez directement les mêmes matrices de poids et vecteurs de *bias* (déjà appris) à ces nouvelles couches.
   Sans relancer d'apprentissage, obtenez les images reconstruites pour tout le *test set* et vérifiez qu'on obtient un résultat équivalent à l'exercice précédent.
6. Fine-tuning : décommentez le bloc de code prévu pour le graphe pluggant une couche *softmax* aux deux couches de codage. Il y a une seule ligne de code à modifier, à savoir l'affectation de la variable `logits`. Pour compléter cette ligne, vous devez juste utiliser le nom que vous avez donné à la deuxième couche de codage dans le graphe complet.
   Décommentez à présent le bloc de code dans la `session` correspondant à l'apprentissage supervisé. Vous n'avez rien à modifier dans ce code. Lancez l'apprentissage complet, et yay ! Le classifieur fonctionne (environ 95% d'*accuracy*), le TP est fini, DatFus est fini... mais DAD continue ! Bon courage pour la dernière ligne droite.