



GL et IA: Apprentissage automatique par analyse de programmes

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr



Références bibliographiques

- Cours NLP de Christopher Manning, Stanford University :
<https://bit.ly/38X1Yeg>
- Article sur **Word2Vec** :
Tomas Mikolov, Kai Chen, Greg Corrado et Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space.*
arXiv :1301.3781v3 [cs.CL] 7 Sep 2013
<https://arxiv.org/pdf/1301.3781.pdf>
- Article sur **Doc2Vec** :
Quoc V. Le, Tomas Mikolov. *Distributed Representations of Sentences and Documents.* arXiv :1405.4053v2 [cs.CL] 22 May 2014 <https://arxiv.org/pdf/1405.4053.pdf>

Références bibliographiques -suite-

- Article sur **Code2Vec** :

Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. *code2vec : Learning Distributed Representations of Code*. Dans les actes de POPL 2019. arXiv :1803.09473v5 [cs.LG] 30 Oct 2018
<https://arxiv.org/pdf/1803.09473.pdf>

- Autres ressources sur Code2Vec :

- Présentation de l'article à POPL'2019 :
<https://www.youtube.com/watch?v=EJ8okcxL2Iw>
 - <https://code2vec.org/>

Plan du cours

1. Introduction rapide au TALN (NLP)
2. Introduction rapide à l'apprentissage automatique
3. TALN & Apprentissage automatique avec Word2Vec
4. Apprentissage automatique par analyse de programmes

Lien IA et TALN

- L'IA (le fait de rendre la machine “intelligente”) a besoin d'un “apprentissage automatisé” (mais pas que !!!)
- L'apprentissage (acquisition de la connaissance) se fait à travers une communication, verbale ou physique (un langage support), à analyser et traiter de façon automatisée
- Intérêt du TALN : étudier la *big picture* des langages humains et la difficulté à les comprendre et les instancier
- Applications : sens des mots/phrases, analyse des dépendances, traduction automatique, réponses automatiques aux questions,
...
- Beaucoup de problèmes complexes à résoudre à cause de la nature ambiguë du langage humain

Sens des mots

Que veut dire “sens” (*meaning*) ? (dictionnaires)

- L'idée représentée par un mot, une phrase, ...
- L'idée qu'une personne veut exprimer en utilisant des mots, des signes, ...
- L'idée exprimée par un travail d'écriture, d'art, ...

Sémantique dénotationnelle :

$$\textit{signifier (symbol)} \Leftrightarrow \textit{signified (idea or thing)}$$

Obtenir le sens d'un mot automatiquement

- Solution possible : utiliser **Wordnet**, thésaurus comportant des ensemble de synonymes (synsets) et hyperonymes

e.g. *synonym sets containing "good"*:

```
from nltk.corpus import wordnet as wn
poses = { 'n':'noun', 'v':'verb', 's':'adj (s)', 'a':'adj', 'r':'adv'}
for synset in wn.synsets("good"):
    print("{}: {}".format(poses[synset.pos()],
                          ", ".join([l.name() for l in synset.lemmas()])))
```

```
noun: good
noun: good, goodness
noun: good, goodness
noun: commodity, trade_good, good
adj: good
adj (sat): full, good
adj: good
adj (sat): estimable, good, honorable, respectable
adj (sat): beneficial, good
adj (sat): good
adj (sat): good, just, upright
...
adverb: well, good
adverb: thoroughly, soundly, good
```

e.g. *hypernyms of "panda"*:

```
from nltk.corpus import wordnet as wn
panda = wn.synset("panda.n.01")
hyper = lambda s: s.hypernyms()
list(pandaclosure(hyper))
```

```
[Synset('procyonid.n.01'),
Synset('carnivore.n.01'),
Synset('placental.n.01'),
Synset('mammal.n.01'),
Synset('vertebrate.n.01'),
Synset('chordate.n.01'),
Synset('animal.n.01'),
Synset('organism.n.01'),
Synset('living_thing.n.01'),
Synset('whole.n.02'),
Synset('object.n.01'),
Synset('physical_entity.n.01'),
Synset('entity.n.01')]
```

Problèmes avec des ressources comme Wordnet

- Ressource intéressante, mais ça manque de nuance
 - *honorable* est listé comme synonyme de *good*, alors que ce n'est valable que dans un certain contexte (lorsqu'on parle d'une personne)
- Les nouveaux mots sont manquants
- Impossible à maintenir à jour (beaucoup de travail manuel nécessaire tout le temps)
- On ne peut pas calculer de façon précise la similarité entre mots

NLP à l'ancienne (2012 et avant)

- On considère les mots comme des symboles discrets (représentation “localiste”)
- Une variable représentant une catégorie de mots (ex : variable hébergement prenant des valeurs comme hôtel, auberge, ...) est représentée par un vecteur :

hôtel = [0 0 0 0 0 1 0 0 0 0]

auberge = [0 0 0 0 0 0 0 0 0 1 0]

- Problème : dimension du vecteur = nombre de mots dans le vocabulaire (très grand)

NLP à l'ancienne (2012 et avant)

- Autre problème : mesure de similarité difficile
 - Exemple : si dans un moteur de recherche, un utilisateur recherche "hôtel à Paris", on aimerait bien faire un match avec les documents contenant "auberge à Paris", sauf que :

hotel = [0 0 0 0 0 0 1 0 0 0 0 0]

auberge = [0 0 0 0 0 0 0 0 0 0 1 0]

 - * Ces deux vecteurs sont orthogonaux
 - * Il n'y a pas de notion naturelle de similarité dans cet encodage
- Autres représentations possibles : Bag of Words, Bag of n-grams, ... mais elles souffrent toutes des mêmes problèmes
- Solution : apprendre à encoder la similarité dans les vecteurs eux-même

Représenter les mots par leur contexte

- Sémantique distributionnelle : Le sens d'un mot est donné par les mots qui apparaissent fréquemment à côté de lui
- L'une des idées les plus réussies du TALN statistique moderne
- Lorsqu'un mot m apparaît dans un texte, son **contexte** est l'ensemble des mots qui apparaissent à côté (dans les limites d'une fenêtre de taille fixe)
- Utiliser plusieurs contextes de m pour construire sa représentation

...government debt problems turning into banking crises as happened in 2009...

...saying that Europe needs unified banking regulation to replace the hodgepodge...

...India has just given its banking system a shot in the arm...

Le sens du mot *banking* = l'ensemble des mots qui l'entourent

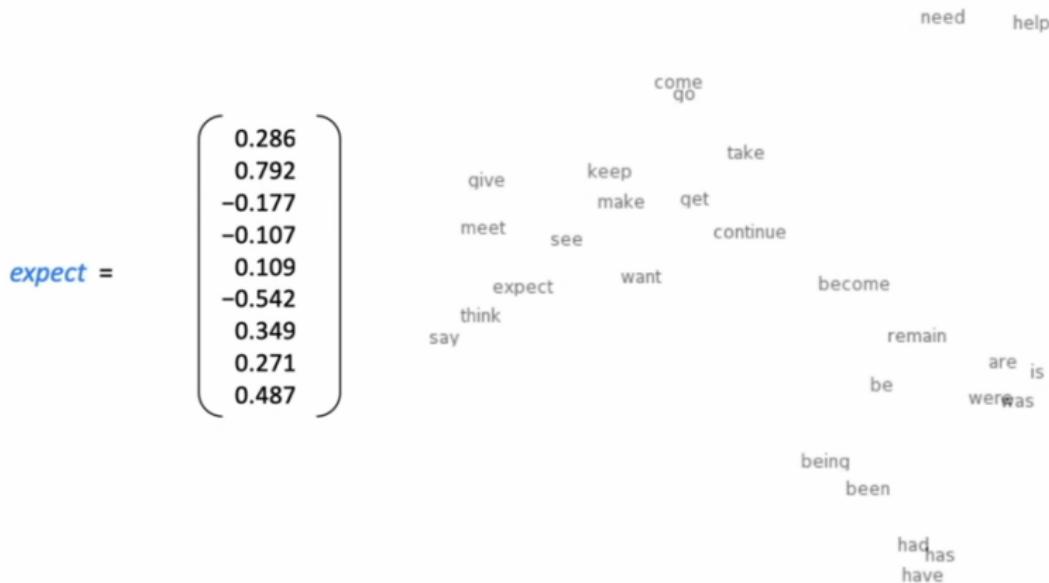
Notion de vecteur de mot (word vector)

- Word vector, appelé également word embedding ou word representation (représentation distributionnelle)
- Un vecteur dense par mot, choisi de telle sorte à ce qu'il soit similaire aux vecteurs des mots qui apparaissent dans le même contexte

$$\text{banking} = \begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \end{pmatrix}$$

- Vecteur de dimension 9 ici, mais souvent, c'est bien plus grand (de l'ordre de plusieurs centaines/milliers, mais ça reste beaucoup plus petit que la dimension des vecteurs vus)

Vecteurs proches



- Cet exemple a été créé avec des vecteurs de dimension 100
- Pour la visualisation, projection dans un espace 2D
- Qu'est-ce qu'on peut observer dessus ?

NLP moderne

- NLP moderne : orientation vers les réseaux de neurones
- A l'origine de cela, l'algorithme Word2Vec, écrit par Tomas Mikolov et al chez Google (voir réfs biblio au début)
- Plusieurs autres implémentations de représentations distributionnelles, mais celle qui a eu le plus d'impact est Word2Vec

Avant de s'intéresser à Word2Vec, un rapide aperçu du ML/DL (*Machine/Deep Learning*) & ANN (*Artifical Neural Networks*)

Plan du cours

1. Introduction rapide au TALN (NLP)
2. **Introduction rapide à l'apprentissage automatique**
3. TALN & Apprentissage automatique avec Word2Vec
4. Apprentissage automatique par analyse de programmes

Problème à résoudre : classification

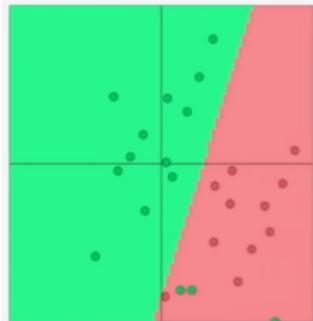
- On a un dataset d'entraînement (*training dataset*) constitué d'échantillons (*samples*)

$$\{x_i, y_i\}_{i=1}^N$$

- x_i sont les entrées, comme par exemple des mots, des phrases, des documents, ...
- y_i sont les sorties : des étiquettes (*labels*), 1 parmi C classes, qu'on veut apprendre à prédire, comme des sentiments (postif ou négatif), thème (topic), d'autres mots, ...

Classification : l'intuition

- Nos x_i sont visualisés en face sur un plan 2D
- Ces points étiquetés (appartiennent à deux classes) : $C = \{\text{rouge}, \text{vert}\}$
- L'objectif de la classification est de déterminer la ligne (fonction) qui délimite les classes (la ligne qui sépare la zone verte de la zone rouge)
- Cas simple ici : vecteurs de mots à 2D et frontière de décision linéaire



Classification : l'intuition -suite-

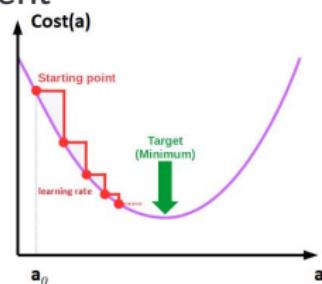
- Méthode : pour tout x , prédire :

$$p(y|x) = \frac{\exp(W_y \cdot x)}{\sum_{c=1}^C \exp(W_c \cdot x)}$$

- L'équation ci-dessus correspond à une fonction connue sous le nom de softmax, qui transforme des nombres quelconques en distribution de probabilités
- L'objectif est d'apprendre (déterminer les meilleures valeurs pour) le vecteur de poids $W \in \mathbb{R}^{C \times d}$

Classification : l'intuition -suite-

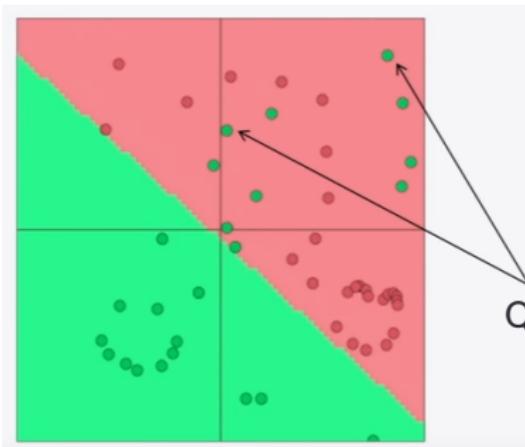
- Problème d'optimisation à résoudre pour maximiser $p(y|x)$ pour l'ensemble d'entraînement
- On peut utiliser la méthode de descente en gradient par exemple pour résoudre ce problème
 - minimiser en réalité $-\log p(y|x)$



- Utiliser ensuite le vecteur de poids W obtenu après entraînement afin de prédire la classe y_i pour un nouvel x_i

Limites de ces méthodes de classification

- Classification linéaire seulement
- Que se passe-t-il avec ça ?



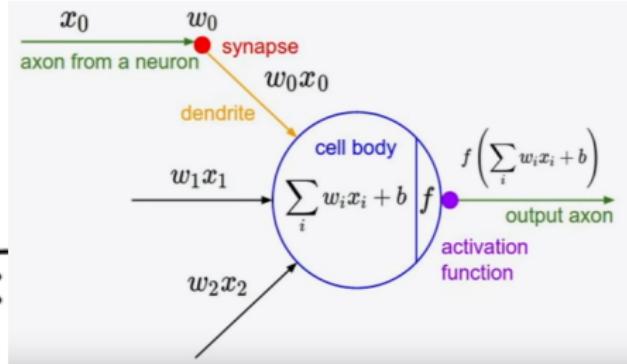
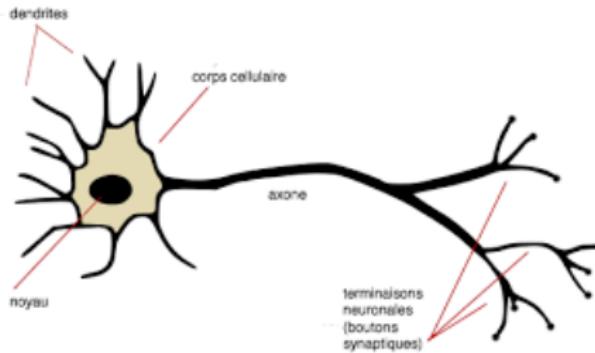
Que faire de ces points ?

Réseaux de neurones, qui viennent au secours

- Les réseaux de neurones sont très adaptés à ce genre de problèmes
- Ils permettent d'obtenir une classification non-linéaire qui peut être très complexe



Neurone humain et neurone artificiel

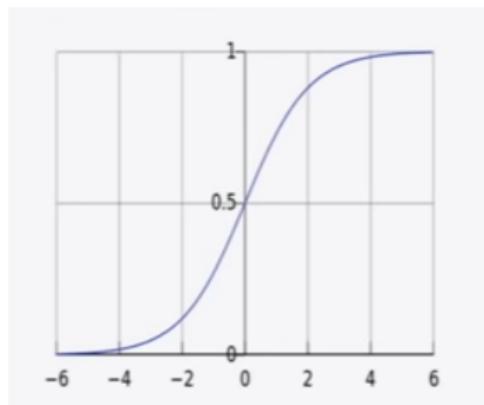


Fonction d'activation

- La fonction d'activation peut être n'importe quelle fonction linéaire ou non
- Typiquement, on utilise une Sigmoïde, comme la régression logistique ou \tanh

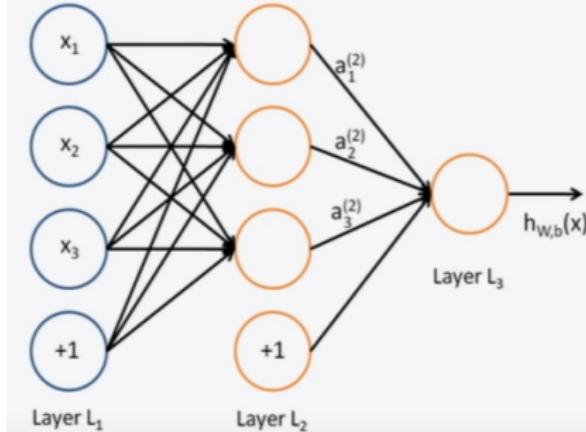
$$h_{w,b}(x) = f(w^T x + b)$$

$$f(z) = \frac{1}{1+e^{-z}}$$



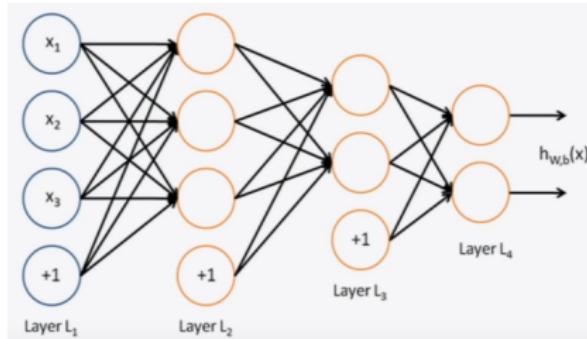
Réseau de neurones

- Un réseau de neurones est un graphe reliant les entrées et sorties à plusieurs neurones pour les exécuter en même temps (plusieurs régressions logistiques simultanément, par ex)
- Réseaux des années 80 et début des années 90 (performances matérielles faibles pour le calcul) = 1 seule couche cachée (L_2)



Réseau de neurones profond

- Un réseau de neurones avec plusieurs couches cachées
- Les couches cachées apprennent des représentations intermédiaires
- Idée inspirée de la vision humaine : de première cellules nerveuses reconnaissent juste les contours, les couleurs, ... avant que les dernières cellules ne reconnaissent l'objet observé par l'œil



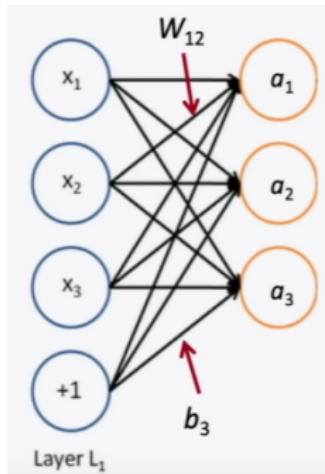
Calcul fait par une couche d'un réseau de neurones

On a :

- $a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$
- $a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$
- ...
- Plus simplement : $z = Wx + b$ et $a = f(z)$

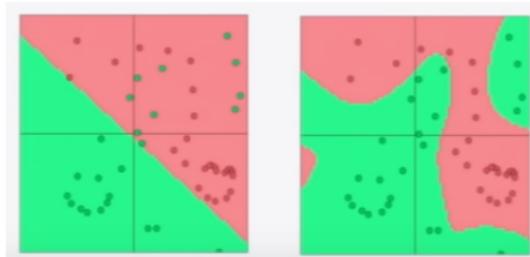
L'activation est faite entrée par entrée :

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$



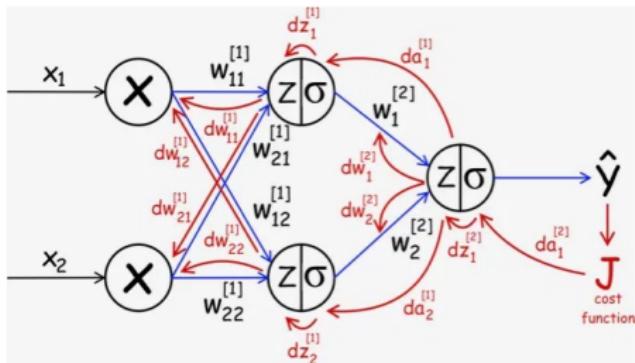
Raisons de la non-linéarité de la fonction d'activation

- Il est très important dans un réseau de neurones profond d'utiliser une fonction d'activation non-linéaire
- Celle-ci permet d'apprendre des formes de courbes complexes pour délimiter la zone de décision (pour la classification)
- Avec des fonctions linéaires cumulées en séquence d'une couche à l'autre, nous obtenons au final une courbe linéaire qui ne nous permet pas d'apprendre des patterns intéressants

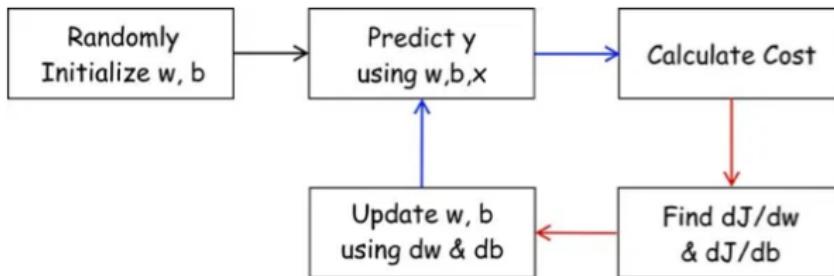


Backpropagation

- Backpropagation = calcul des gradients de façon algorithmique et efficace pour propager l'optimisation de l'erreur (de combien les poids W doivent être mis à jour) dans le sens opposé du calcul vu précédemment



Algorithme de calcul



- Nombreux calculs complexes à faire si réseaux profond (dW et db)
- Optimisation : en utilisant la programmation dynamique, certains calculs sont factorisés
- Calculs proposés par de nombreux frameworks d'apprentissage profond (TensorFlow, PyTorch, ...)
 - Pas besoin de les implémenter, dans la plupart des cas

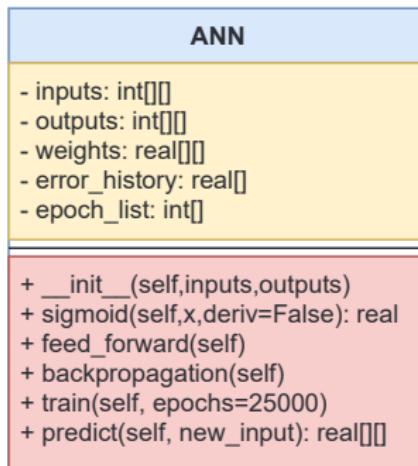
Différence avec l'apprentissage automatique (DL vs ML)

Apprentissage profond (*Deep Learning*)

Avec les réseaux de neurones profonds, les features sont appris de façon automatique, au lieu de requérir cela en entrée de l'outil de classification (pas de *feature engineering* en amont)

Exemple (tiré de <https://bit.ly/3ojog30>)

- Une classe Python ANN (Artificial Neural Network) représentant un réseau de neurones simple (pas profond)



Exemple -le programme principal-

```
# input data
inputs = np.array([[0, 1, 0],[0, 1, 1],[0, 0, 0],
                  [1, 0, 0],[1, 1, 1],[1, 0, 1]])
# output data
outputs = np.array([[0], [0], [0], [1], [1], [1]])
# create neural network
NN = ANN(inputs, outputs)
# train neural network
NN.train()
# create two new examples to predict
example_1 = np.array([[1, 1, 0]])
example_2 = np.array([[0, 1, 1]])
# print the predictions for both examples
print(NN.predict(example_1),
      ' - Correct: ', example_1[0][0])
print(NN.predict(example_2),
      ' - Correct: ', example_2[0][0])
# ...
```

Exemple -le programme principal -suite-

```
# ...
# plot the error over the entire training duration
plt.figure(figsize=(15,5))
plt.plot(NN.epoch_list, NN.error_history)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.show()
```

Exemple -la classe ANN-

```
class ANN:

    # initialize variables in class
    def __init__(self, inputs, outputs):
        self.inputs = inputs
        self.outputs = outputs
    # initialize weights as .50 for simplicity
    self.weights = np.array([[.50], [.50], [.50]])
    self.error_history = []
    self.epoch_list = []

#activation function ==> S(x) = 1/1+e^(-x)
def sigmoid(self, x, deriv=False):
    if deriv == True:
        return self.sigmoid(x, deriv=False) * (1 - self.sigmoid(x, deriv=False))
    return 1 / (1 + np.exp(-x))
```

Exemple -la classe ANN -suite-

```
# Suite de la classe ANN:

# data will flow through the neural network.
def feed_forward(self):
    self.hidden = self.sigmoid(np.dot(self.inputs, self.
weights))

# going backwards through the network to update weights
def backpropagation(self):
    self.error = self.outputs - self.hidden
    delta = self.error * self.sigmoid(self.hidden, deriv=
True)
    self.weights += np.dot(self.inputs.T, delta)
```

Exemple -la classe ANN -suite-

```
# Suite de la classe ANN:  
# train the neural net for 25,000 iterations  
def train(self, epochs=25000):  
    for epoch in range(epochs):  
        # flow forward and produce an output  
        self.feed_forward()  
        # go back though the network to make corrections based on the output  
        self.backpropagation()  
        # keep track of the error history over each epoch  
        self.error_history.append(np.average(np.abs(self.error)))  
        self.epoch_list.append(epoch)  
  
    # function to predict output on new and unseen input data  
def predict(self, new_input):  
    prediction = self.sigmoid(np.dot(new_input, self.weights))  
    return prediction
```

Code complet sur Moodle. Le tester sur vos machines

Exemple -suite-

- A votre avis, qu'est-ce que le réseau a appris (quel feature) ?
- Remplacer la valeur de `example_2` par
`np.array([[0, 0, 1]])` et retester
- Que remarquez vous ?
- C'est un exemple qui ne marche pas bien, parce que le réseau n'a pas été entraîné avec suffisamment de données
- L'un des "défauts" des réseaux de neurones artificiels : il leur faut beaucoup de données d'entraînement et il faut beaucoup de diversité dans ces données
- Ce sont des outils de classification qui s'apprêtent bien aux domaines où les données existent en grand nombre
(multimédia : images et voix, texte,... et code source ?!!!)

Plan du cours

1. Introduction rapide au TALN (NLP)
2. Introduction rapide à l'apprentissage automatique
3. TALN & Apprentissage automatique avec Word2Vec
4. Apprentissage automatique par analyse de programmes

Word2Vec

C'est un framework pour l'apprentissage de vecteurs de mots

L'idée est très simple :

- Il faut disposer d'un large corpus de texte
- Chaque mot dans un vocabulaire restreint est représenté par un vecteur (aléatoire au départ)
- Aller à chaque position t dans le texte, qui a un mot central c et des mots de contexte (*outside*) o
- Calculer la probabilité de o étant donné c (similarité = produit de leurs vecteurs), ou inversement
- Ajuster les vecteurs continuellement (des millions de fois!!!) pour maximiser cette probabilité

Variantes du modèle word2vec

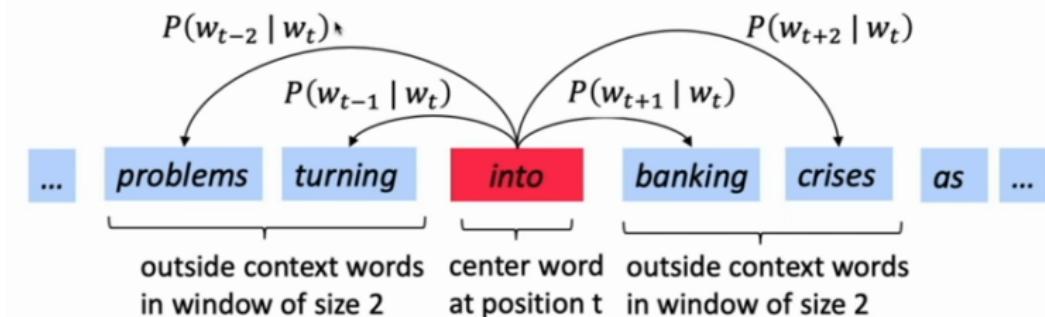
Il existe deux variantes :

- Skip-grams (SG) : prédire les mots du contexte (*outside*) étant donné un mot central (le modèle présenté précédemment)
- Continuous Bag-of-words (CBOW) : prédire un mot central étant donné des mots du contexte

L'article de Mikolov et al. présente également les optimisations à faire dans le calcul des vecteurs pour passer à l'échelle (*negative sampling*)

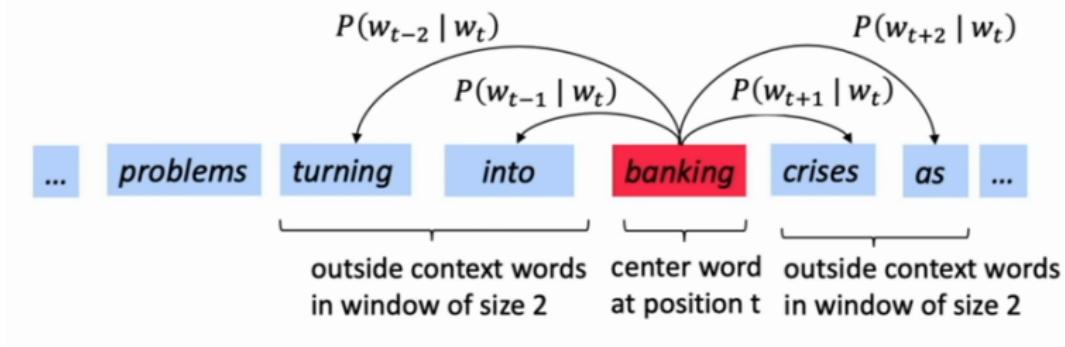
Exemple de fenêtre dans le modèle Skip-grams

- Calculer $P(w_{t+j} | w_t)$: (identifier une seule distribution de probabilités)



Exemple de fenêtre -mot suivant-

- Maximiser cette probabilité $P(w_{t+j} | w_t)$ (sminimiser la fonction de perte : $1 - P(w_{t+j} | w_t)$) :



Fonction d'objectif (*objective function*)

- Pour chaque position $t = 1, \dots, T$, dans le texte, prédire les mots du contexte dans une fenêtre de taille fixe m , étant donné un mot central w_t :

$$\text{Likelihood} = L(\theta) = \prod_{t=1}^T \prod_{\substack{-m \leq j \leq m \\ j \neq 0}} P(w_{t+j}|w_t; \theta)$$

Likelihood = vraisemblance

- La fonction d'objectif (la fonction de perte ou de coût, *cost/loss function*) $J(\theta)$ est la moyenne négative du log de $L(\theta)$

$$J(\theta) = -\frac{1}{T} \log(L(\theta)) = -\frac{1}{T} \sum_{t=1}^T \sum_{\substack{-m \leq j \leq m \\ j \neq 0}} \log P(w_{t+j}|w_t; \theta)$$

Minimiser la fonction d'objectif \Leftrightarrow maximiser la précision de la prédiction

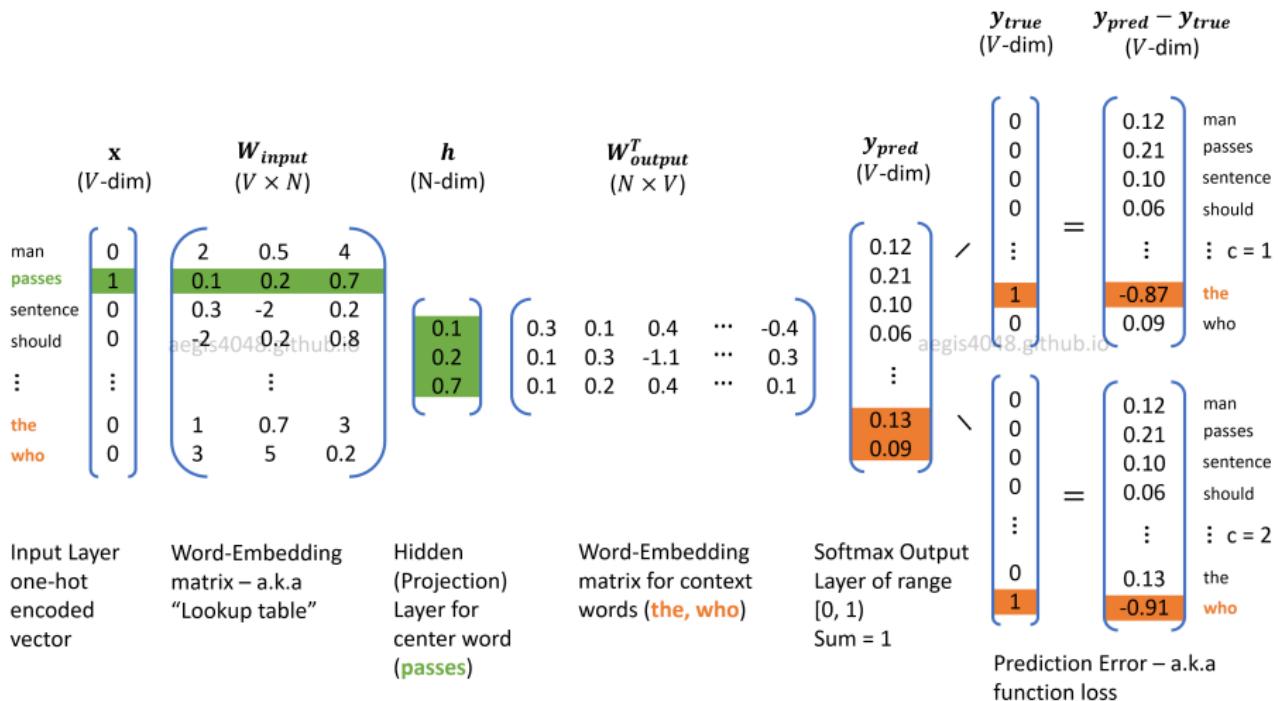
Comment calculer $P(w_{t+j}|w_t; \theta)$? (de façon simple)

- On utilise deux vecteurs par mot :
 - v_w quand le mot w est un mot central
 - u_w quand le mot w est un mot de contexte
- Pour un mot central c et un mot de contexte o :

$$P(o|c) = \frac{\exp(u_o^t v_c)}{\sum_{w \in V} \exp(u_w^t v_c)}$$

- Cette équation est équivalente à la fonction softmax (généralisation de la Sigmoïde), avec :
 - un numérateur qui est l'exponentielle (pour que le résultat soit positif, et donner de l'importance aux grandes valeurs) du produit vectoriel (qui permet de mesurer la similarité) u_o^t et v_c
 - un dénominateur qui est la somme des exponentielles pour tout le vocabulaire (pour rapporter le tout à une mesure de proba)

Architecture du modèle Skip-grams



Ajuster les vecteurs

- Les vecteurs sont modifiées de façon répétitives afin d'optimiser la fonction d'objectif
- Apprentissage (des valeurs des vecteurs – word embeddings) en minimisant la fonction de perte $L(\theta)$
- On calcule les gradients de tous les vecteurs (1 mot = 2 vecteurs)

$$\theta = \begin{bmatrix} v_{aardvark} \\ v_a \\ \vdots \\ v_{zebra} \\ u_{aardvark} \\ u_a \\ \vdots \\ u_{zebra} \end{bmatrix} \in \mathbb{R}^{2dV}$$

Exemple d'utilisation de Word2Vec

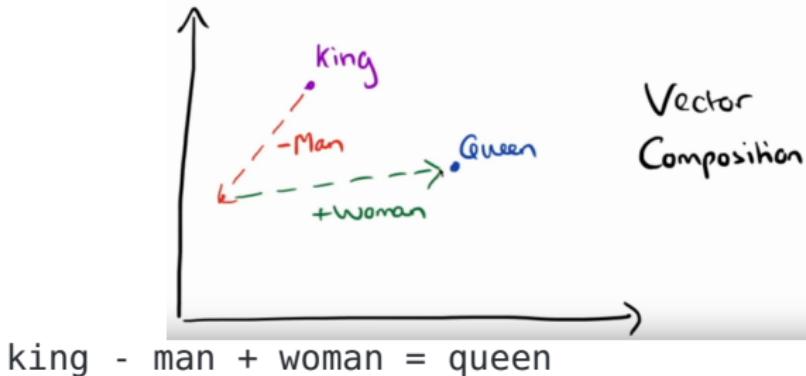
- Avec Gensim (<https://radimrehurek.com/gensim/>)

```
import numpy as np
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
print(wv.most_similar(positive=['car', 'minivan'],
                      topn=5))
# OUT: [('SUV', 0.8532192707061768), ('vehicle',
0.8175783753395081), ('pickup_truck',
0.7763688564300537), ('Jeep', 0.7567334175109863),
('Ford_Explorer', 0.7565720081329346)]
print(wv.doesnt_match(['fire', 'water', 'land', 'sea',
                      'air', 'car']))
# OUT: car
```

- A tester

Autre exemple intéressant

- Composition de vecteurs (analogies entre mots)



```
print(wv.most_similar(positive=['woman', 'king'],
                      negative='man')[0])
print(wv.most_similar(positive=['france', 'german'],
                      negative='germany')[0])
print(wv.most_similar(positive=['france', 'beer'],
                      negative='germany')[0])
```

Et doc2vec ? (Le and Mikolov, 2014 – voir réfs biblio)

- Une extension de word2vec pour l'analyse de documents contenant du texte (au delà d'un mot)
- Lors de l'apprentissage, un autre vecteur (vecteur de paragraphe/document) est calculé et optimisé
- Ce vecteur donne une représentation numérique du paragraphe/document
- Très utile dans la recommandation de documents ou l'analyse des sentiments (positifs, négatifs ou neutre, leur intensité, ...) dans les tweets, avis de consommateurs, ...
- Deux implém : *Paragraph Vector - Distributed Memory (PV-DM)* & *Paragraph Vector - Distributed Bag of Words (PV-DBOW)*

Plan du cours

1. Introduction rapide au TALN (NLP)
2. Introduction rapide à l'apprentissage automatique
3. TALN & Apprentissage automatique avec Word2Vec
4. Apprentissage automatique par analyse de programmes

Quel intérêt ?

- Synthèse de programmes, auto-complétion intelligente, ...
- Ex : étiquetage (labellisation) sémantique des programmes

```
String[] _____(final String[] array) {  
    final String[] newArray = new String[array.length];  
    for (int index = 0; index < array.length; index++) {  
        newArray[array.length - index - 1] = array[index];  
    }  
    return newArray;  
}
```

reverseArray

77.34%

reverse

18.18%

subArray

1.45%

Quel intérêt ? -suite-

```
boolean f(Object target) {  
    for (Object elem: this.elements) {  
        if (elem.equals(target)) {  
            ① return true;  
        }  
    }  
    ② return false;  
}
```

(a)

Predictions:

| | | |
|---------------|--|--------|
| contains | | 90.93% |
| matches | | 3.54% |
| canHandle | | 1.15% |
| equals | | 0.87% |
| containsExact | | 0.77% |

```
Object f(int target) {  
    for (Object elem: this.elements) {  
        if (elem.hashCode().equals(target)) {  
            ① return elem;  
        }  
    }  
    ② return this.defaultValue;  
}
```

(b)

Predictions

| | | |
|-------------|--|--------|
| get | | 31.09% |
| getProperty | | 20.25% |
| getValue | | 14.34% |
| getElement | | 14.00% |
| getObject | | 6.05% |

```
int f(Object target) {  
    int i = 0;  
    for (Object elem: this.elements) {  
        if (elem.equals(target)) {  
            ① return i;  
        }  
        i++;  
    }  
    ② return -1;  
}
```

(c)

Predictions

| | |
|---------------------|--|
| indexOf | |
| getIndex | |
| findIndex | |
| indexOfNull | |
| getInstructionIndex | |

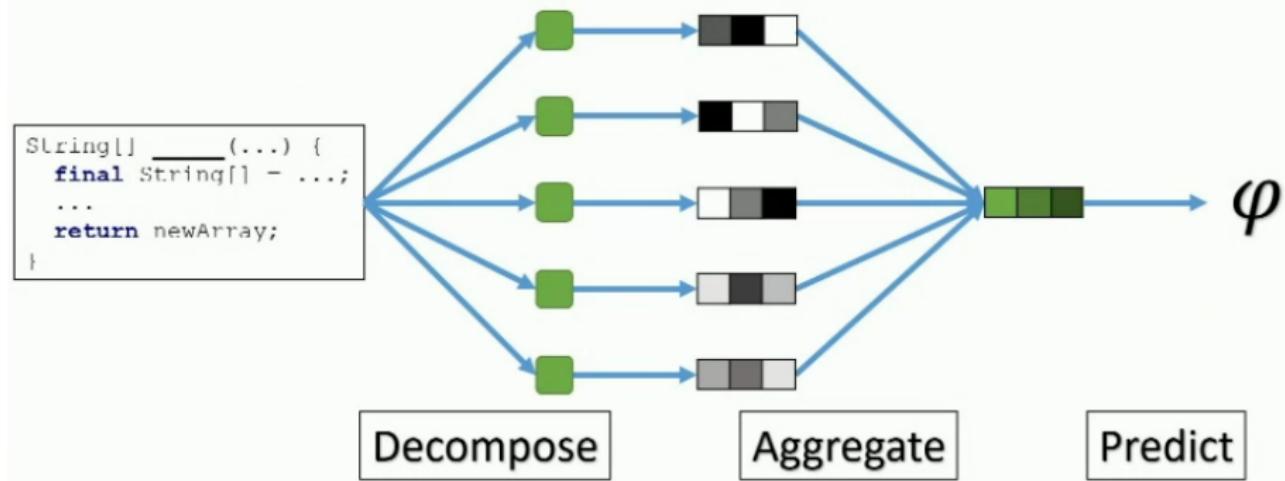
L'idée

- Utiliser les programmes (des millions de programmes) comme données en entrée à un algorithme d'apprentissage (un réseau de neurones artificiels, par ex)
- L'algorithme va apprendre des propriétés sur ces programmes : par exemple comment nommer les méthodes dans les classes
- Une fois qu'on a le modèle entraîné, on l'utilise pour prédire une propriété sur un nouveau programme (trouver le nom d'une nouvelle méthode)
- Exemple d'algorithme : Code2vec → un réseau de neurones pour prédire les propriétés du code
- Autres propriétés pouvant être apprises : présence ou non de malwares, dépendances requises, mots clés/hashtags, détection de clones, ...

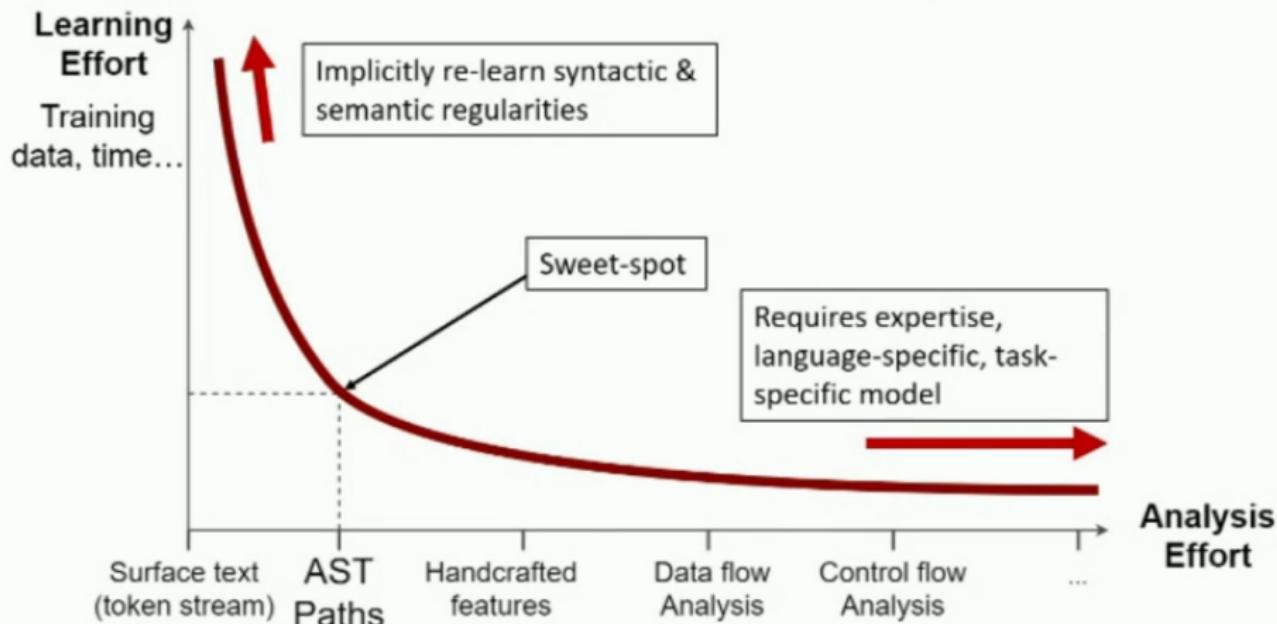
Challenges pour l'analyse des programmes

- Comment décomposer les programmes en plus petits **blocs** ?
 - des blocs suffisamment petits pour reproduire ça sur un ensemble de programmes
 - mais suffisamment grands pour qu'ils soient significatifs (parlants)
- Comment ensuite agréger un ensemble de ces blocs pour donner une seule prédiction ?

Schéma général



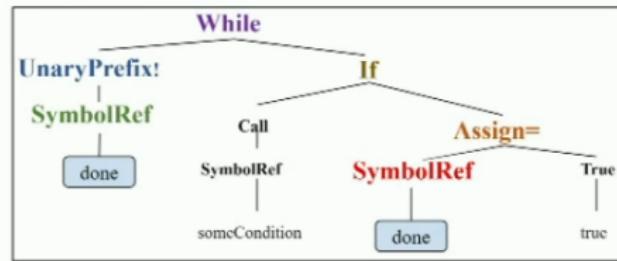
Challenge #1 : décomposition de programmes



AST Paths = bon compromis, parce qu'ils encodent la syntaxe,
mais pas toute la sémantique

AST Paths

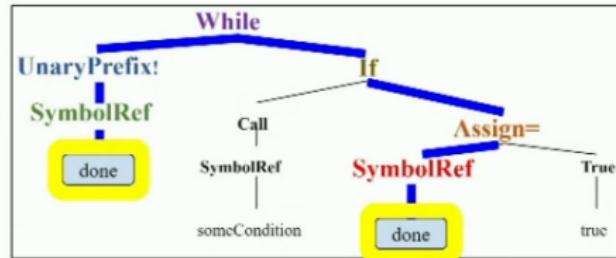
```
while (!done) {  
    if (someCondition()) {  
        done = true;  
    }  
}
```



- Un programme = un ensemble composé de tous les chemins (AST Paths) entre nœuds feuilles de l'arbre syntaxique (AST)

AST Paths -suite-

```
while (!done) {  
    if (someCondition()) {  
        done = true;  
    }  
}
```



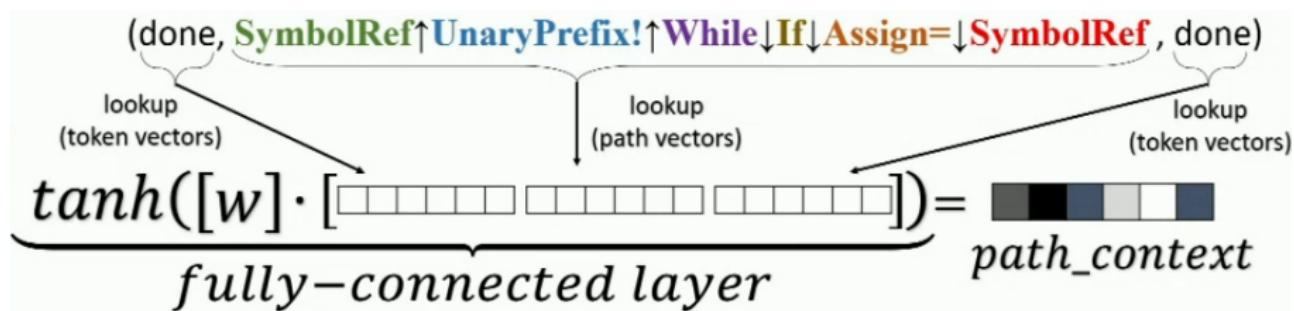
(done, **SymbolRef** ↑ **UnaryPrefix!** ↑ **While** ↓ **If** ↓ **Assign=** ↓ **SymbolRef** , done)

- Un AST path capture une partie de la sémantique du programme : variable done est utilisée dans la condition d'une boucle while (tant qu'elle est fausse), elle sera initialisée à true dans un if

Représentation des AST Paths comme vecteurs

Deux sortes de vecteurs sont entraînés/utilisés :

- Vecteurs de jetons (*Token Vectors*)
- Vecteurs de chemins (*Path Vectors*)



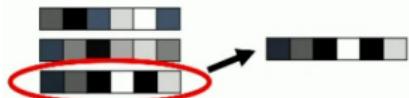
Il faudra maintenant agréger ces multiples vecteurs (un par chemin) dans un seul vecteur de code

Challenge #2 : agrégation des chemins

Trois possibilités

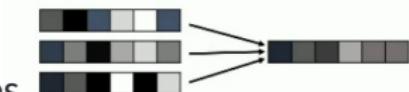
1. Utiliser le vecteur le plus important

- Perte d'informations utiles encodées dans les autres vecteurs



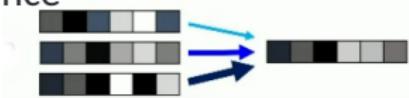
2. Fusionner les vecteurs

- Donner un poids égal à des informations qui ne sont pas forcément aussi importantes les unes que les autres



3. Privilégier le vecteur qui a le plus d'importance

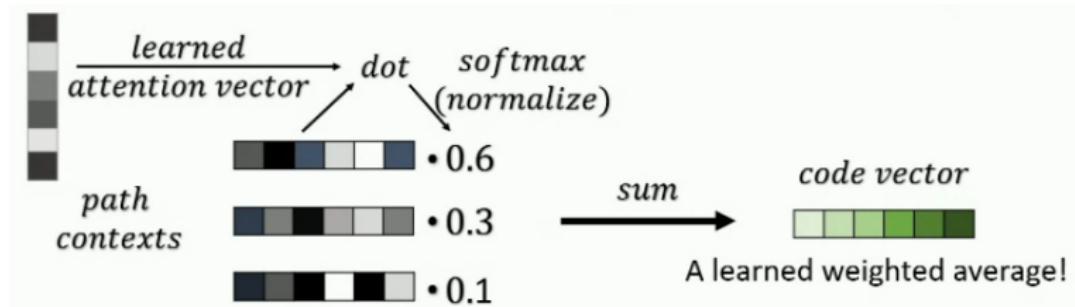
Attention = moyenne pondérée apprise



Notion d'Attention (neuronale)

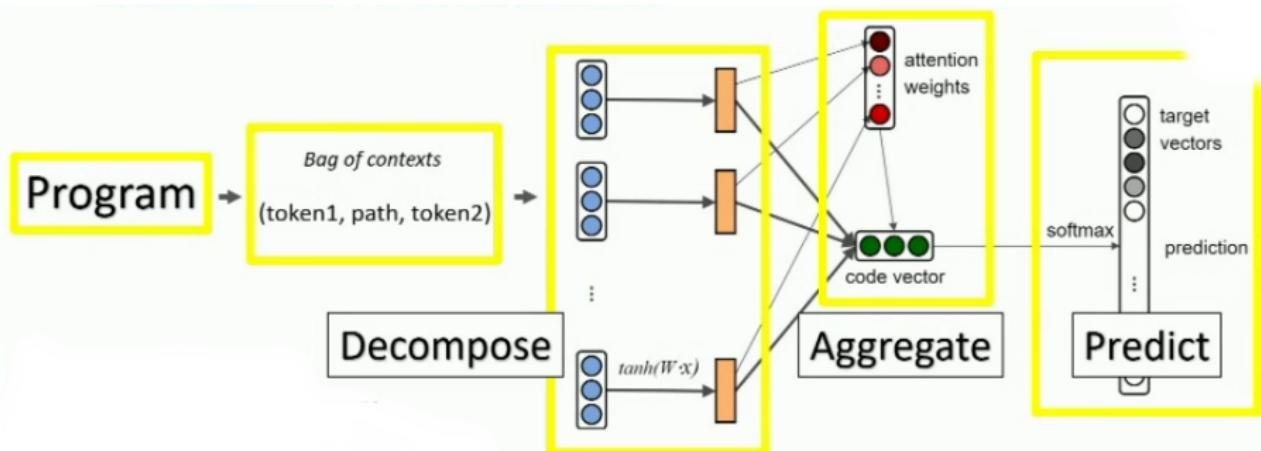
L'idée de base consiste à ce que les vecteurs apprennent deux choses distinctes :

1. La sémantique d'un chemin (*AST Path*)
2. Le niveau d'attention que ce chemin doit avoir



Le vecteur d'attention est initialisé aléatoirement au départ, il est mis à jour et entraîné avec le reste du réseau de neurones

Architecture complète de code2vec



- Le réseau est entraîné entièrement en même temps
- Exemple : prédiction des noms de méthodes (training set = 14M d'exemples - méthodes, durée d'entraînement = < 1 jour)

Exemple

```
boolean f(Object target) {  
    for (Object elem: this.elements) {  
        if (elem.equals(target)) {  
            return true;  
        }  
    }  
    return false;  
}
```

Predictions:

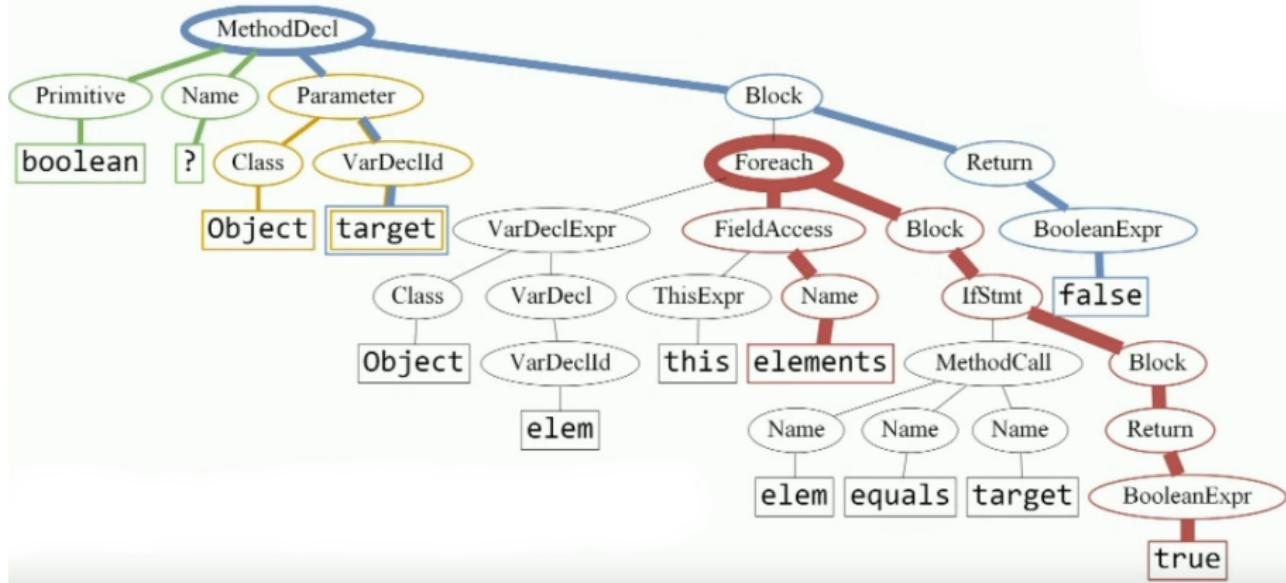
| | | |
|----------------------|--|--------|
| contains | | 90.93% |
| matches | | 3.54% |
| canHandle | | 1.15% |
| equals | | 0.87% |
| containsExact | | 0.77% |

```
Object f(int target) {  
    for (Object elem: this.elements) {  
        if (elem.hashCode().equals(target)) {  
            return elem;  
        }  
    }  
    return this.defaultValue;  
}
```

Predictions

| | | |
|--------------------|--|--------|
| get | | 31.09% |
| getProperty | | 20.25% |
| getValue | | 14.34% |
| getElement | | 14.00% |
| getObject | | 6.05% |

Exemple -suite-



L'attention permet l'interprétabilité des résultats (gros problème en DL)

Autres usages

A tester sur : <http://code2vec.org>

- Un nom de méthode est similaire à ? Tester : count
- Combinaisons de noms : Tester equals and toLower
- Analogies : Tester receive is to download as send is to ...
- Prédire le nom d'une méthode : tester les exemples donnés

Tester également vos propres méthodes pour voir quels noms l'outil prédit

Exercice

- Aller sur le site <https://github.com/tech-srl/code2vec>
- Cloner le projet Git (Step 0 sur le site)
- Télécharger le modèle code2vec de taille 1.4GB (Step 2)
- Réaliser ce qui est indiqué dans la Step 4, en éditant le fichier Input.java avec vos propres exemples et observer les AST paths et leurs poids
- Trouver un moyen de récupérer de façon programmatique les context-paths et leurs poids (*attention vector*)
- Est-ce pertinent d'utiliser ces paths pour identifier des bad-smells si par exemple il n'y a pas de poids dominant (méthode avec plusieurs fonctionnalités) ? Tester sur des exemples

De la lecture

- C. Watson, N. Cooper, D. N. Palacio, K. Moran et D. Poshyvanyk. **A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research.** ACM TOSEM. Septembre 2021 <https://arxiv.org/abs/2009.06520>
- Revue systématique de la littérature : 128 articles sélectionnés (de bons pointeurs !) sur les 10 ans sur le thème GL et DL
- Taxonomie basée sur : 1) quelles tâches de GL ?, 2) quelles méthodes de DL (réseaux de neurones spécifiques) ont été utilisées ?, 2) sur quels artefacts ?, ...
- Tâches de GL traitées : Source Code Generation, Code Comprehension, Source Code Retrieval & Traceability, Bug-Fixing Processes, Feature Location, Clone Detection, ...

De la lecture sur l'IA, au delà du GL

