**Les réseaux de neurones avec TensorFlow !**

**Comment est-ce qu’il se construit et s’optimise ?**

1. **Construction**
   1. **Les données d’entrainement**

On se sert de placeholder pour définir la structure de nos tensors :

*input\_image = tf.placeholder(tf.string)*

* 1. **L’inférence**



L’inférence représente toutes les étapes dont nous avons besoin afin de réaliser des prédictions des classes à partir des images de notre set d’images. C’est là que nous construisons l’architecture de notre réseau de neurone.

TensorFlow nous permet à cette étape pour chaque image de calculer un vecteur de probabilités. Pour une Softmax Regression cela donne :

with tf.name\_scope("softmax"):

poids = tf.Variable(tf.zeros([num\_pixels, num\_classes]))

biais = tf.Variable(tf.zeros([num\_classes]))

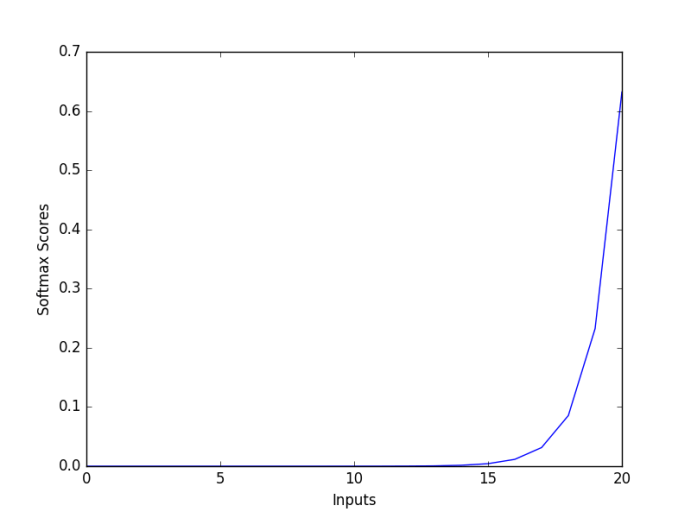
logits = tf.matmul(x, weights) + biases

softmax = tf.nn.softmax(logits)

On note que si les valeurs des poids et des biais sont hautes, les neurones auront tendance à apprendre la même chose…

Pour visualiser la variation de valeur du softmax, tensorFlow nous fournis un histogramme :

tf.summary.histogram(softmax.op.name + '/activations', softmax)



* 1. **La fonction de coût**

Pour que notre réseau de neurone obtienne des résultats cohérents, il nous faut définir une fonction de coût que l’on optimise à chaque nouvelle itération. Celle-ci sert donc à diriger l’apprentissage du réseau pendant laquelle on cherche à minimiser l’erreur de prédiction.

L’entropie croisée (cross-entropie) est généralement utilisée comme fonction de coût. Elle a la particularité de mesurer le nombre de bits moyen dont nous avons besoin pour identifier un évènement (tags) issue d’un ensemble d’évènement (dictionnaire)

Pour faire simple, elle nous permettra de mesurer à quel point les prédictions que nous auront calculées sont éloignées de la vérité. Les poids et les biais que nous avons vu plus haut seront donc mis à jour grâce à cette fonction :

with slim.arg\_scope(inception.inception\_v3\_arg\_scope()):

logits, end\_points = inception.inception\_v3(

traiter\_image, num\_classes=FLAGS.num\_classes, is\_training=False)

* 1. **Entrainement**

Etape d’optimisation de notre réseau de neurone. On procède en affectant des règles sur l’attribution des poids de notre système (A CHAQUE ITERATION).

On s’appuie sur la méthode de rétropropagation du gradient qui calcule le gradient de l’erreur pour chaque neurone de notre système (sur toute les couches) combiné avec des algorithmes par exemple la descente du gradient stochastique.

C’est donc cette erreur que nous cherchons à minimiser !

Pour cela, il nous faut définir la vitesse d’entrainement de notre système que l’on appellera Learning Rate. Cette variable impacte la variation des poids et des biais de notre réseau de neurone.

Selon les valeurs que prendront cette variable, la convergence de notre système se verra différente.

Avec un Learning Rate élevé, on observe de nette variations des poids et biais quand l’erreur est jugée importante. Point négatif, il ne faut pas que cette valeur soit trop élevé sinon notre système diverge.

descenteGradienOptim = tf.train.GradientDescentOptimizer(learning\_rate)

* 1. **Visualisation des résultats grâce à TensorBoard**

**A FAIRE :**

* [**https://blog.xebia.fr/2017/03/22/tensorflow-deep-learning-episode-2-notre-premier-reseau-de-neurones/**](https://blog.xebia.fr/2017/03/22/tensorflow-deep-learning-episode-2-notre-premier-reseau-de-neurones/)
* [**https://www.tensorflow.org/programmers\_guide/summaries\_and\_tensorboard**](https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard)

*Proposer une liste d’image en lien avec une autre sélectionner au préalable :*

* *Classification d’images automatique*
  + *Réseau de neurone « TensorFlow »*
    - *Plus de 200.000 images pour l’entrainement*
    - *Softmax Regression*