# 1 Introduction

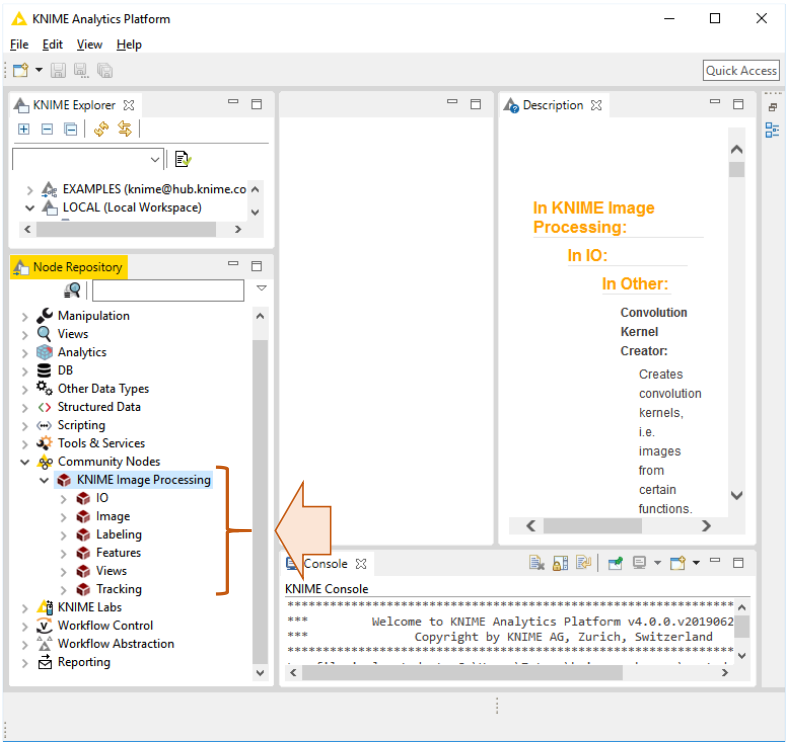
**Implémentation d'un réseau de neurones à convolution (CNN – Convolutional Neural Networks) pour la catégorisation d'images sous KNIME. Utilisation des composants deep learning de Keras (Tensorflow backend).**

Ecrire un tutoriel sur l'utilisation des réseaux de neurones convolutifs (CNN) pour le classement d'images me titillait depuis un moment déjà. Mais il y a tellement de choses à écrire que je repoussais sans cesse. La lecture récente du dernier ouvrage de Stéphane Tufféry (Tufféry, 2019) et la découverte des composants de deep learning sous KNIME m'ont poussé à me lancer.

Il existe de nombreux didacticiels en ligne, notamment sur l'utilisation des CNN en Python sur des bases qui font référence telles que MNIST ou ''Cats and Dogs''. Stéphane dans son ouvrage effectue les mêmes analyses, mais sous R. Ça ne sert à rien de les réitérer. Mon idée était de me démarquer en proposant une étude simplifiée sur une base moins usitée, en schématisant les étapes autant que possible, et en réalisant l'ensemble des traitements sans écrire une seule ligne de code. Varier les plaisirs ne peut pas faire de mal. KNIME convient parfaitement dans ce cadre.

# 2 Installation des outils

Pour l'installation des composants de Deep Learning Keras sous KNIME, je conseille mon précédent tutoriel « Deep Learning avec Keras sous KNIME » (juillet 2019, **section 2**). Nous nous assurons de surcroît que les composants de traitement d'images ont bien été intégrés.

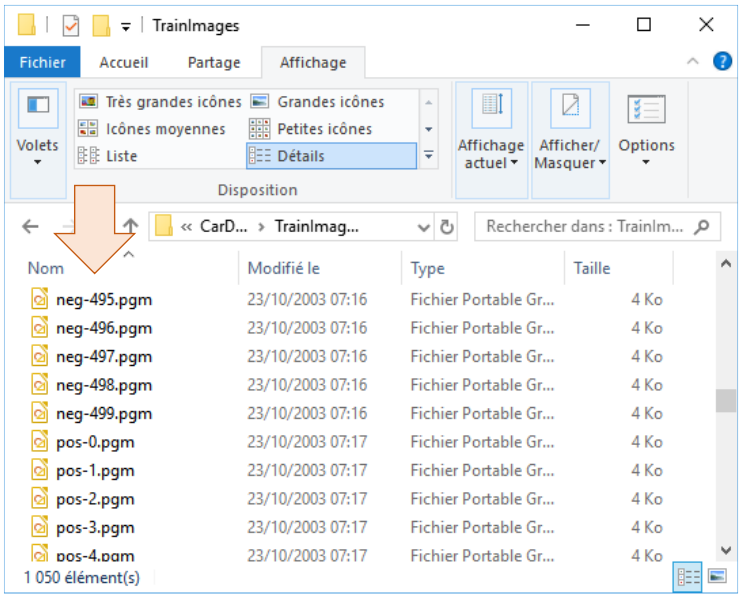


# 3 Données

## 3.1 Source et description

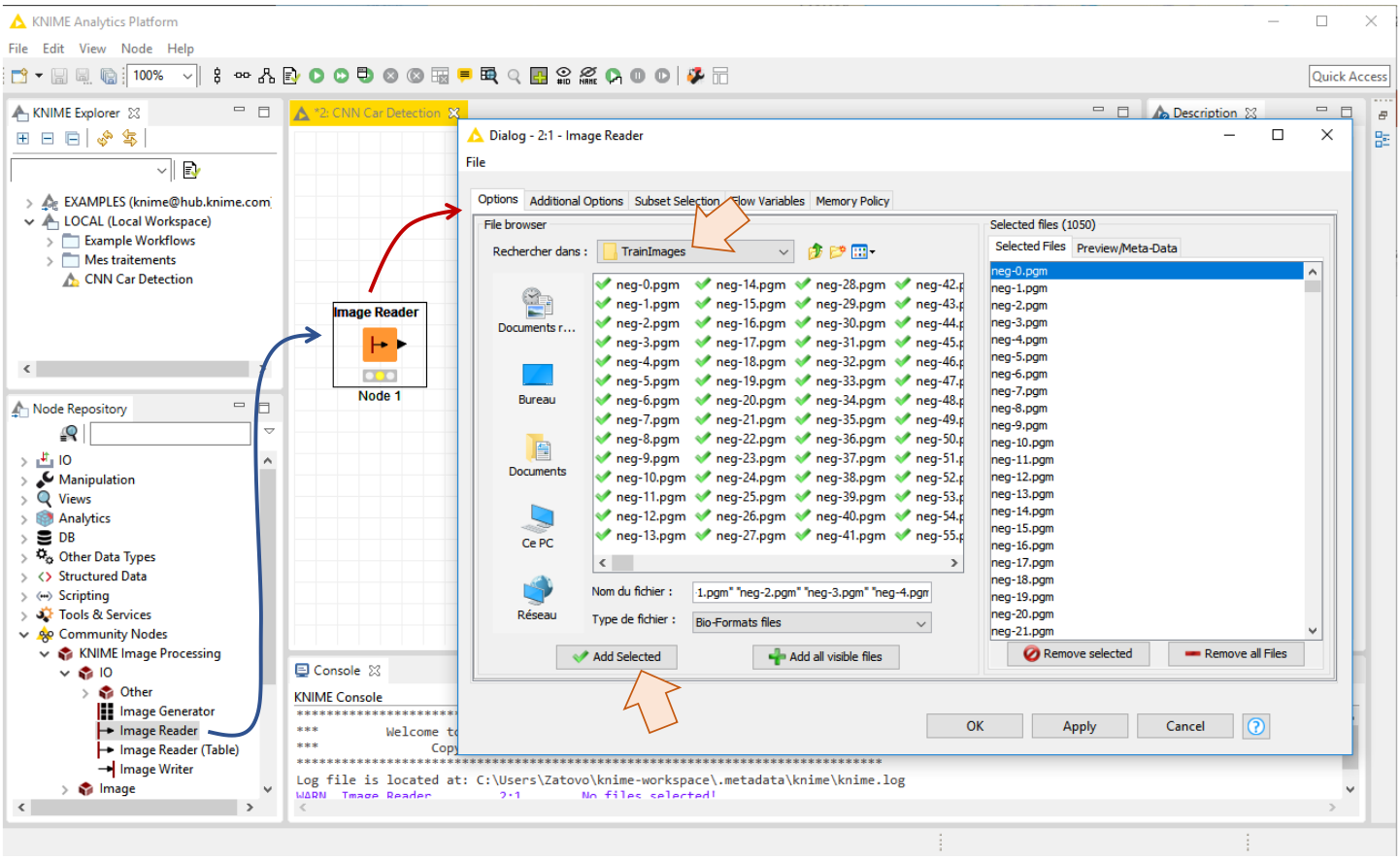
La base ''UIUC Image Database for Car Detection'' contient des photos au format PGM de véhicules vus de profil (observations ''positives'') et de tout autre objet (observations ''négatives''). L'objectif est de distinguer les images de véhicules des autres. Nous avions déjà utilisé cette base dans un précédent tutoriel (''Image mining avec Knime'', juin 2016). Nous nous focalisons sur l'ensemble ''TrainImages'' comportant 1050 photos que nous scinderons en échantillon d'apprentissage (750) et de test (300).

Les images sont regroupées dans un dossier. Les trois premières lettres des noms de fichiers permettent d'identifier la classe (positif ou négatif). Il faudra les décortiquer pour créer la variable cible binaire (pos vs. neg) nécessaire aux traitements.

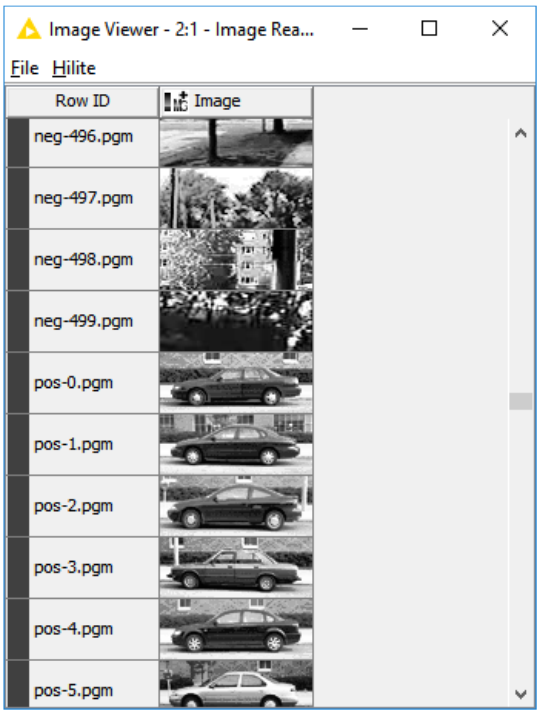



## 3.2 Importation

Nous créons un workflow sous Knime (voir ''Analyse prédictive sous Knime'', février 2016). Nous le nommons ''CNN Car Detection''. Nous importons les images avec le composant **IMAGE READER**. Nous le paramétrons (menu CONFIGURE) pour désigner les fichiers du dossier ''TrainImages''. Nous disposons de 1050 observations.



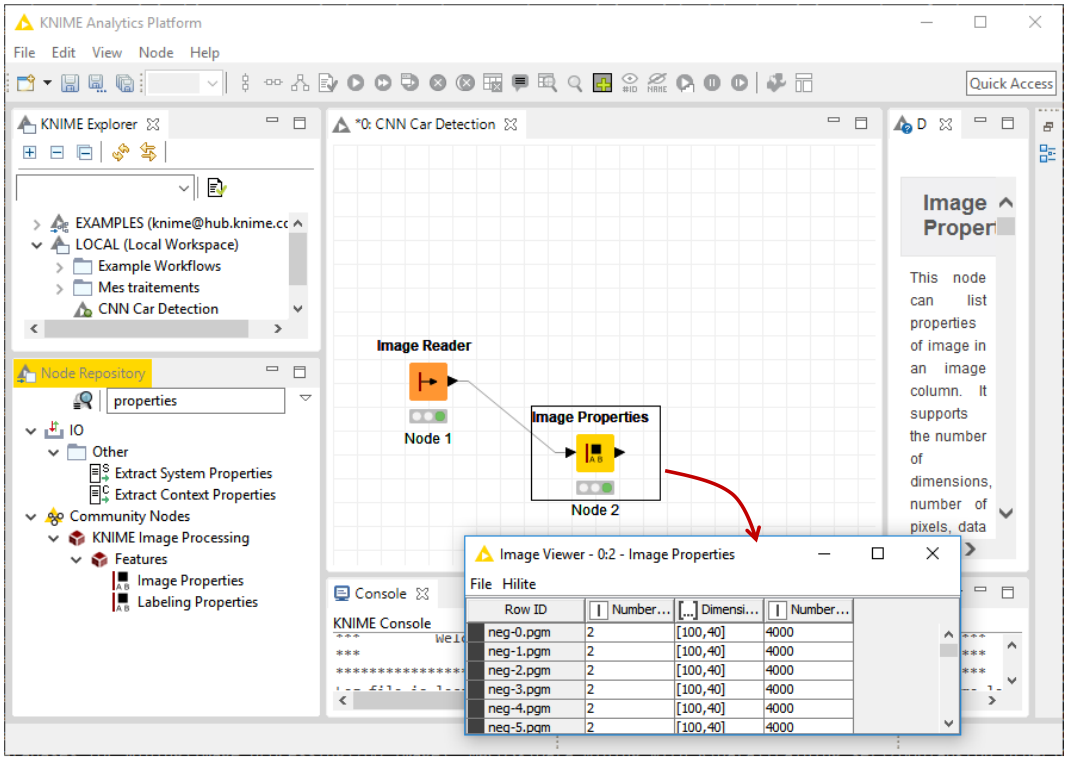

Nous pouvons visualiser les images en cliquant le menu EXECUTE AND OPEN VIEWS.




Première information, les images sont en niveau de gris.

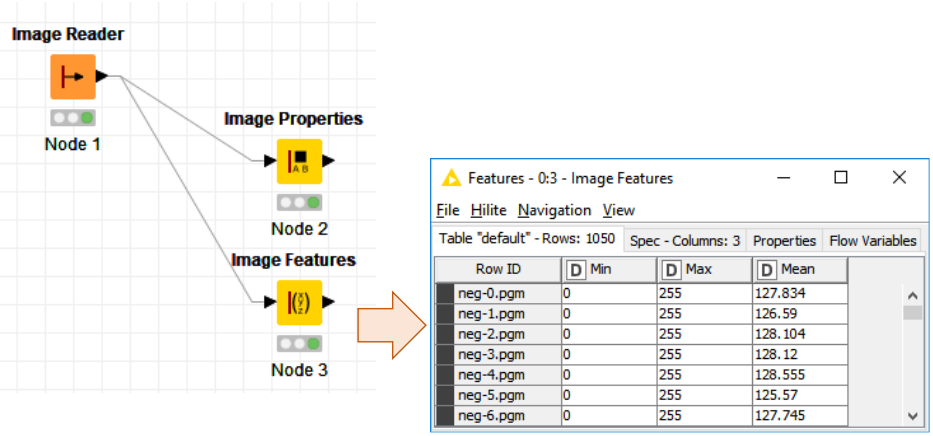
## 3.3 Inspection des caractéristiques

Connaître les caractéristiques des images est très important pour pouvoir leur appliquer les traitements qui conviennent. Nous utilisons le composant **IMAGE PROPERTIES** pour inspecter leurs propriétés. Dans la boîte de paramétrage (menu CONFIGURE), nous allons sur l'onglet FEATURES et nous sélectionnons : Number of Dimensions, Dimensions, Number of Pixels. Nous lançons les calculs avec EXECUTE AND OPEN VIEWS.




Les photos sont en 2 dimensions, de taille (100 x 40) avec une définition de 4000 pixels.

Elles sont en niveau de gris, avec un seul canal donc. Vérifions les plages de valeurs prises par les pixels avec le composant **IMAGE FEATURES**. Dans la boîte de paramétrage, nous optons pour FIRST ORDER STATISTICS dans l'onglet FEATURES avec : Min, Max, Mean.

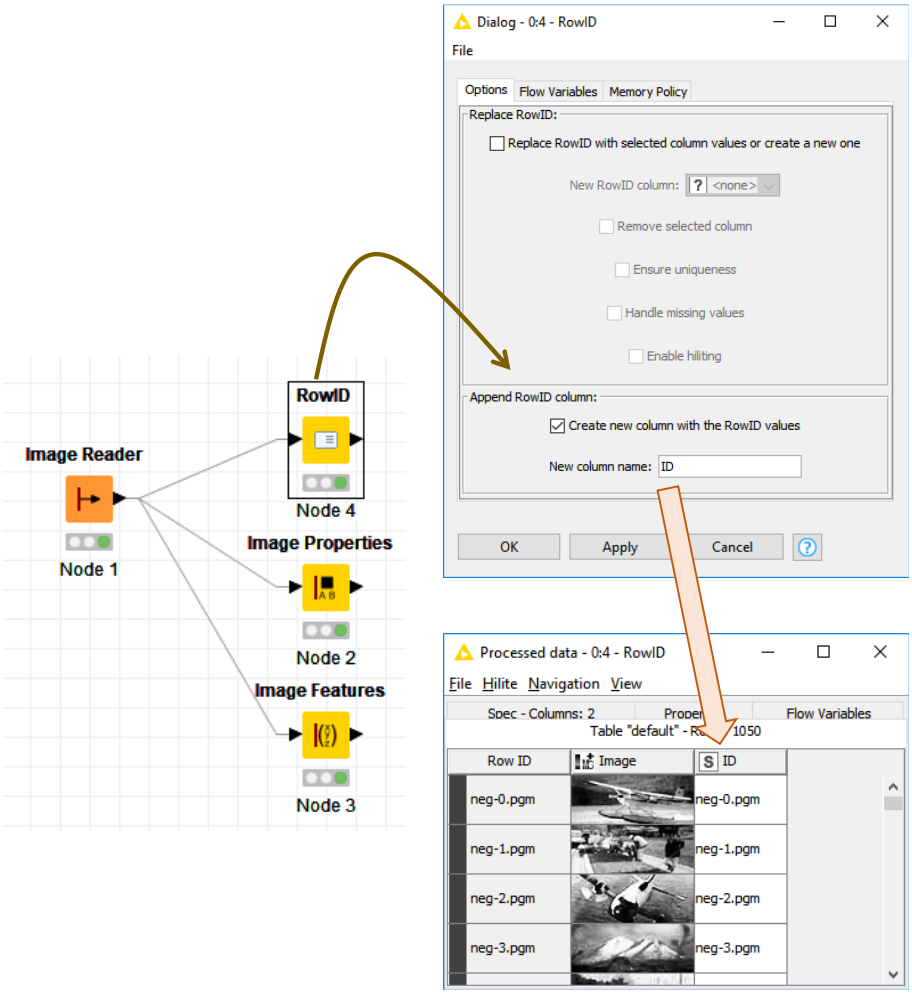


Les valeurs sont bien comprises entre 0 et 255.

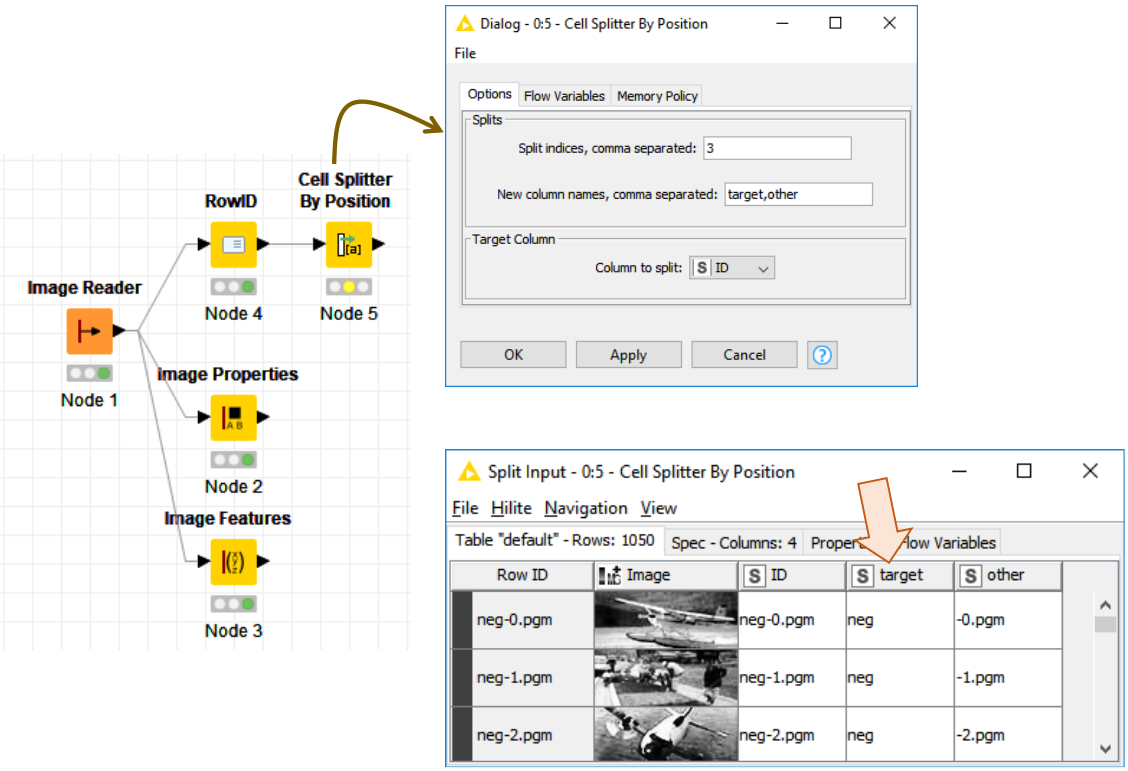
Remarque : Une stratégie possible à ce stade consisterait à normaliser ces valeurs en 0 et 1 pour mieux guider l'apprentissage statistique par la suite (Tufféry, 2019 ; section 5.3). Nous passons outre néanmoins parce que nous opterons pour une autre stratégie de normalisation plus loin.

## 3.4 Création de la variable cible

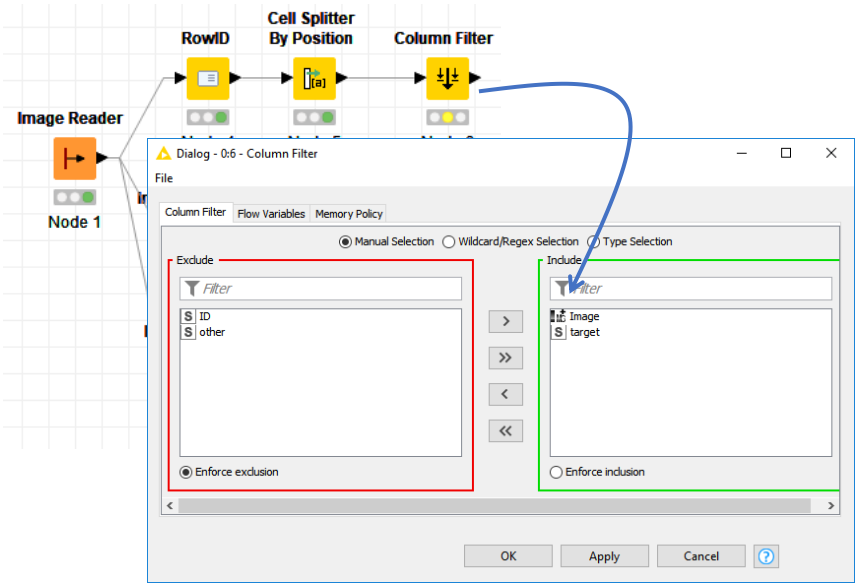
Nous cherchons à extraire les étiquettes (pos vs. neg) des observations à partir des 3 premiers caractères des noms des fichiers (voir ''Image mining avec Knime'', juin 2016 ; section 3.4 pour une présentation plus détaillée). Pour ce faire, nous utilisons **ROW ID** dans un premier temps. Nous le paramétrons pour qu'il déduise une nouvelle variable, que nous nommerons ''ID'', à partir de ces identifiants.




Dans un second temps, avec **CELL SPLITTER BY POSITION**, nous extrayons les 3 premiers caractères de ''ID'' pour en déduire la variable ''TARGET''.

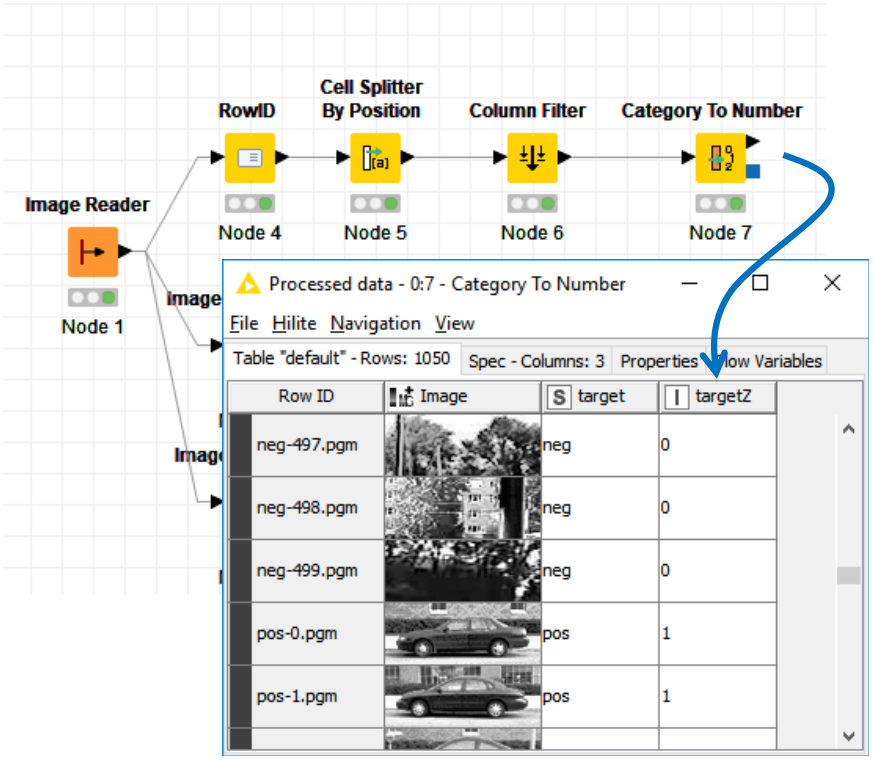



Nous procédons enfin à un filtrage des colonnes pour ne conserver que les images et la variable cible pour la suite des opérations. Nous utilisons **COLUMN FILTER**.




## 3.5 Recodage de la cible

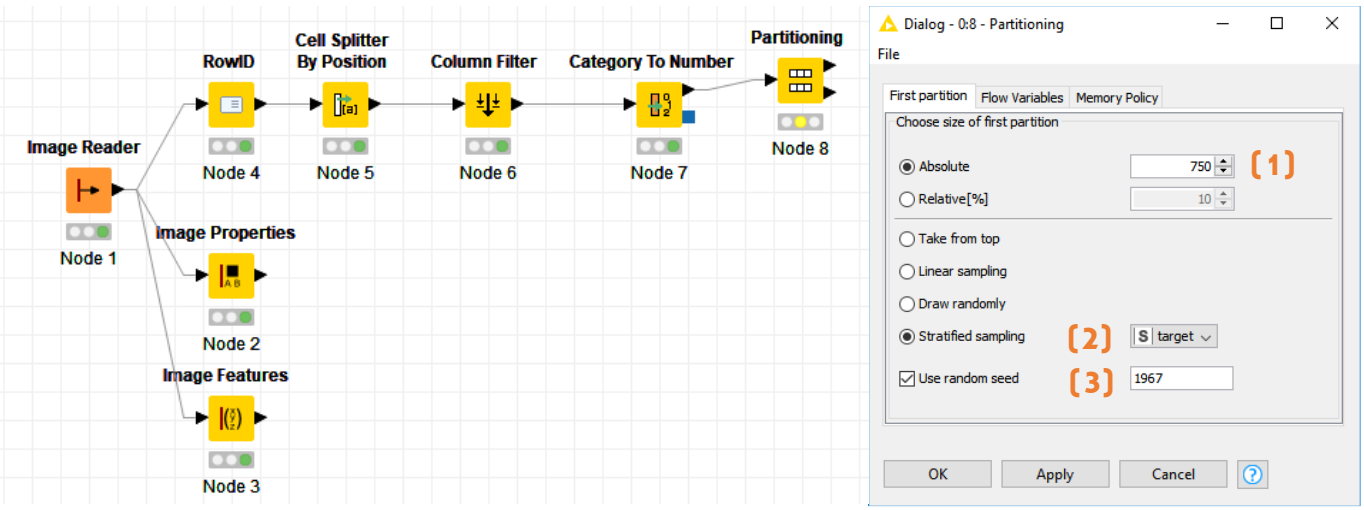
La variable cible est de type chaîne de caractères, symbolisée par la lettre ''S'' dans la copie d'écran ci-dessus. Keras a besoin qu'elle soit recodée en numérique pour réaliser la modélisation. Nous utilisons le composant CATEGORY TO NUMBER (voir ''Deep Learning avec Keras sous Knime'', juillet 2019 ; section 4.2 pour le paramétrage).




Notre nouvelle variable s'appelle ''**targetZ**''.

## 3.6 Subdivision apprentissage-test

Puisque nous sommes dans un schéma supervisé, nous scindons les données en échantillons d'apprentissage (750 observations) **(1)** et de test (300) avec l'outil **PARTITIONING**. Nous optons pour un tirage stratifié pour que les proportions des classes soient respectées dans les deux sous-ensembles **(2)**. Nous fixons SEED à 1967 pour que l'expérimentation soit reproductible à l'identique **(3)**.



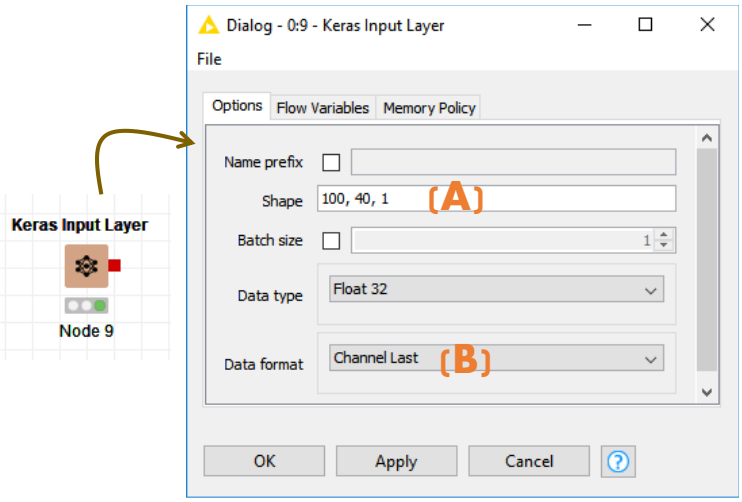

Nous sommes maintenant parés pour lancer la modélisation.

# 4 Implémentation d'un CNN avec Keras sous KNIME

Je vais à l'essentiel dans cette section. Pour une meilleure compréhension des réseaux de neurones convolutifs, je conseille (voir la bibliographie) la lecture de l'ouvrage de Stéphane Tufféry (2019, section 5.5 et suivantes). L'article de Dumoulin et Visin (2016), pour les opérateurs (convolution, pooling) que nous mettons à contribution dans ce tutoriel tout du moins, me paraît très intéressant également.

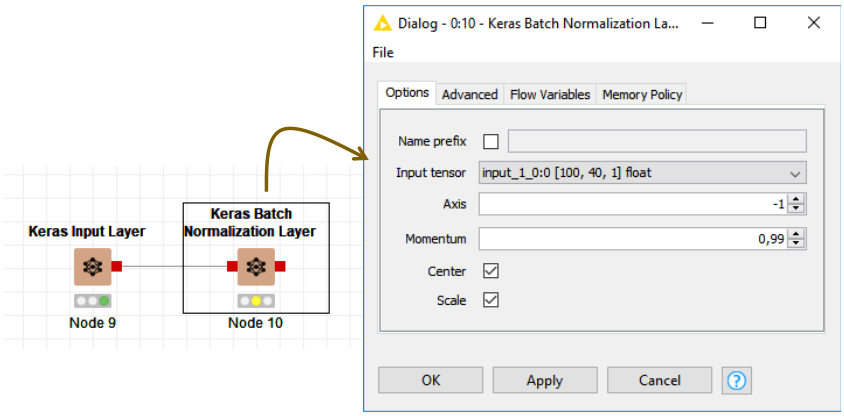
## 4.1 Définition du réseau

**Couche d'entrée**. Notre réseau est composé de plusieurs couches. Nous commençons naturellement par la couche d'entrée **KERAS INPUT LAYER** branchée sur les matrices d'images. Nous le paramétrons comme suit :

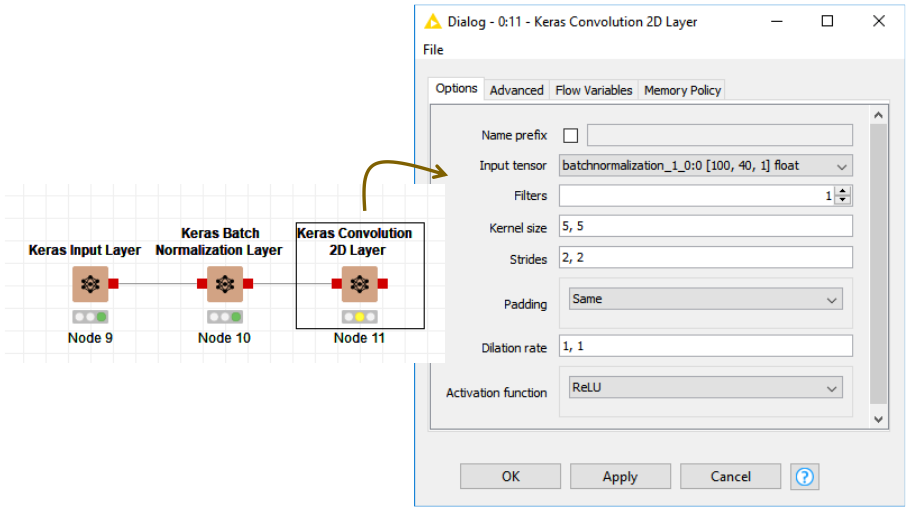



Le paramètre SHAPE permet de spécifier la dimension (**100, 40**, 1) des images **(A)**. La dernière valeur représente le nombre de canaux. Nous n'en avons qu'un seul puisque nous sommes en niveaux de gris. Nous indiquons **(B)** qu'il est en dernière position dans le triplet de SHAPE.

**Normalisation par lot**. La couche de normalisation par lot est usuellement utilisée pour stabiliser la distribution des données au cours de l'apprentissage (Tufféry, 2019 ; section 5.8). Pour le groupe d'observations constituant le lot (''batch'', nous y reviendrons lors du paramétrage du KERAS NETWORK LEARNER, section 4.2), les valeurs des matrices représentant les images sont centrées et réduites. Placé à cet endroit du réseau, on peut le voir surtout comme une alternative à la normalisation des images d'entrées que nous n'avions pas réalisé en amont lors de la première appréhension des données. Nous introduisons le composant **KERAS BATCH NORMALIZATION LAYER** dans le workflow. Le paramétrage par défaut n'a pas été modifié. Il a su détecter les dimensions des images lors de la connexion avec la couche précédente.



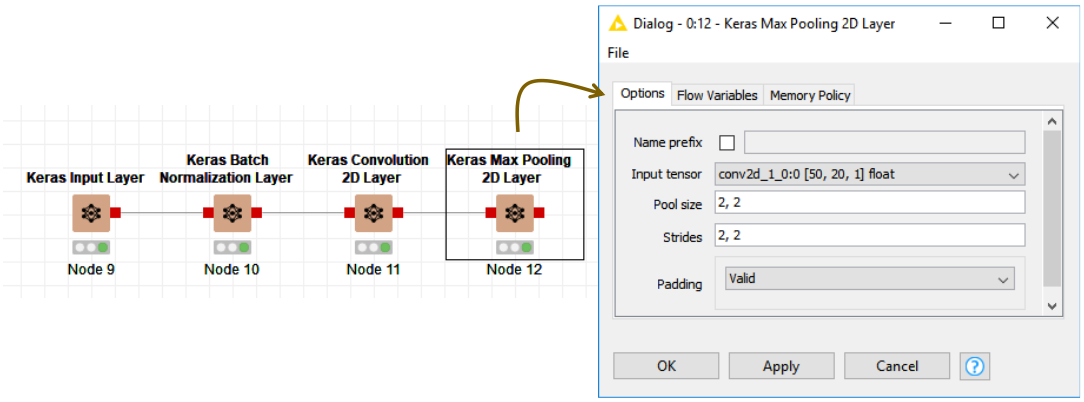

**Couche de convolution 1**. La couche de convolution permet d'extraire les caractéristiques visuelles (Tufféry, 2019 ; section 5.6). Avec une fenêtre glissante appelée noyau, elle effectue un SOMMEPROD avec la portion traitée de la matrice représentant l'image (Dumoulin et Visin, 2016 ; section 1.1). Avec **KERAS CONVOLUTION 2D LAYER** (2D parce que nous traitons des images représentées par des matrices), dans l'onglet OPTIONS, nous optons pour (5, 5) pour la taille de la fenêtre (**Kernel Size**), avec un décalage (**Strides**) de (2, 2) dans les deux sens (respectivement en ligne et colonne) à chaque étape. Nous remplissons l'espace avec des zéros pour couvrir complètement l'image (**Padding = Same**) (voir Dumoulin et Visin, 2016 ; Figure 1.2). Nous optons pour une fonction d'activation (**Activation function**) ReLU, très populaire en traitement d'images (Tufféry, 2019 ; page 93). Enfin, dans l'onglet ADVANCED, nous désactivons le biais (colonne de valeurs 1 représentant la constante qui ne s'impose pas vraiment dans une couche de convolution) en désélectionnant **Use bias?**



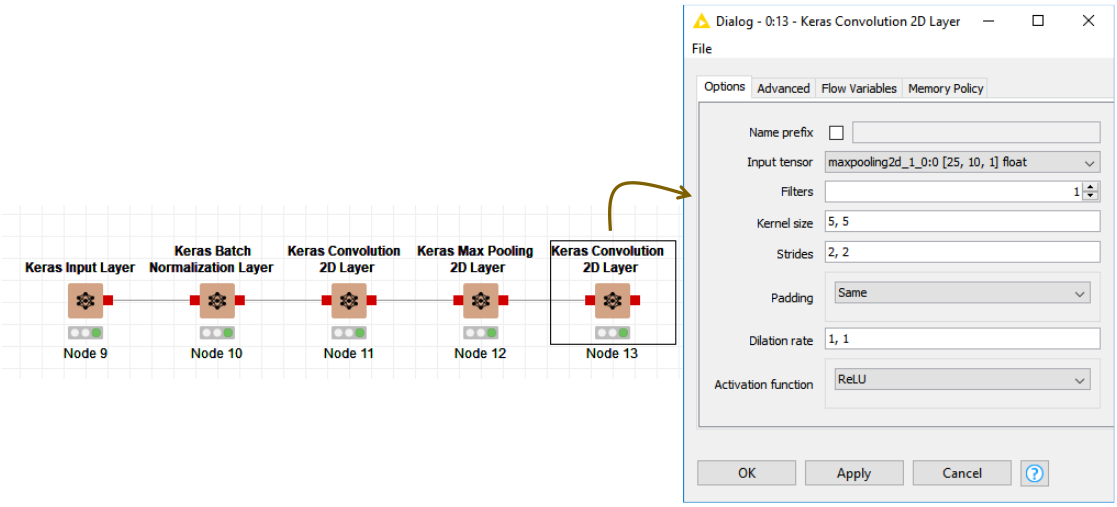

Avec ces paramètres (Kernel, Strides, Padding), la taille de la matrice est divisée par 2 en sortie de la couche. Elle est de taille (**50, 20**).

**Couche de Max Pooling**. Les zones traitées par la convolution se recouvrent avec notre paramétrage de la couche précédente. Pour réduire la redondance, nous utilisons le pooling

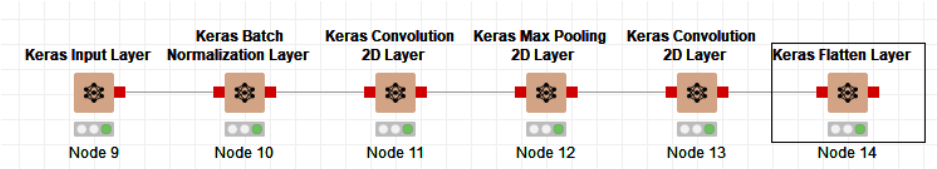
(Tufféry, 2019 ; section 5.9). **KERAS MAX POOLING 2D LAYER** prend la valeur maximale pour chaque bloc dont la taille est fixée à (2, 2) (**Pool size = 2, 2**). Le pas de déplacement (**Strides**) est (2, 2). De fait, nous réduisons par 2 encore la dimension des matrices, soit (**25, 10**).




**Couche de convolution 2**. Nous ajoutons une seconde couche de convolution pour extraire d'autres caractéristiques des images, déjà bien transformée à ce stade. Vraiment pour simplifier parce qu'il n'y a pas de raisons objectives pour qu'il en soit ainsi, nous spécifions les mêmes paramètres que précédemment (Kernel Size, Strides, Padding, Activation function, Use bias). La dimension de la matrice est une nouvelle fois divisée par deux, soit (**13, 5**).

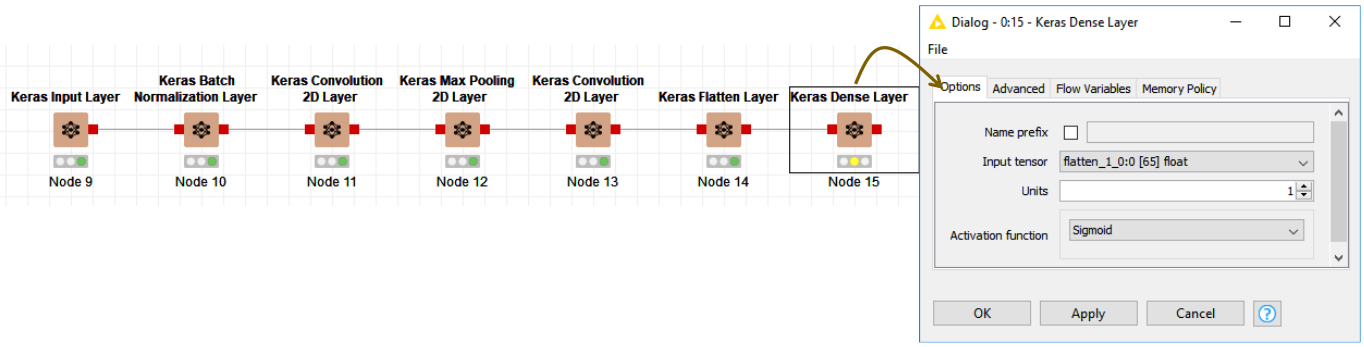



**Couche Flatten**. L'aplatissement (Tufféry, 2019 ; Figure 6.3) consiste à linéariser la structure matricielle. D'une matrice de dimension (13, 5), nous en déduisons un vecteur unidimensionnel de dimension (13 x 5 = 65). Cette opération est nécessaire pour initier la partie ''perceptron'' du réseau. Nous introduisons le composant **KERAS FLATTEN LAYER**.



**Couche dense**. Puisque nous optons pour un perceptron simple, cette couche dense **KERAS DENSE LAYER** correspond aussi à la couche de sortie de notre réseau. Nous avons fait le choix de la simplicité pour cet exemple didactique. Mais rien ne vous empêche de mettre plusieurs couches denses pour élaborer un perceptron multicouche plus sophistiqué. Avec peut-être l'opportunité de proposer un modèle plus puissant, mais avec aussi le désavantage du risque de surapprentissage.

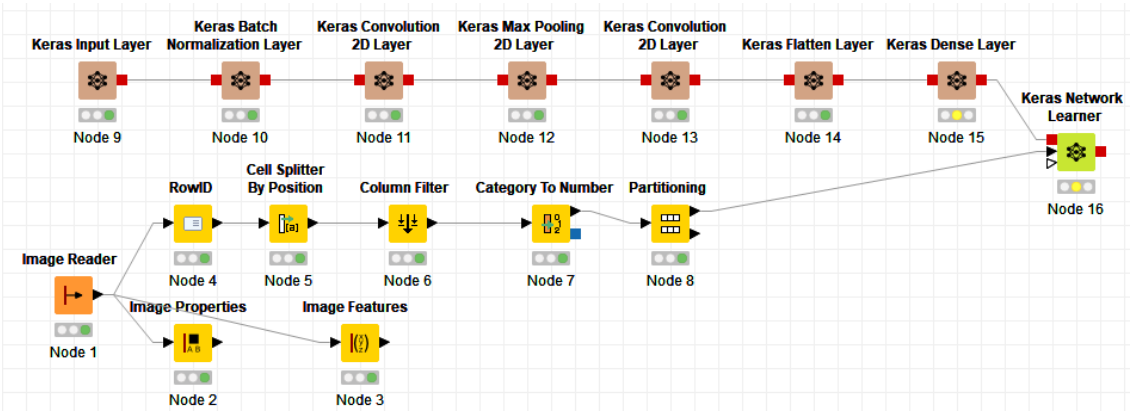
La variable cible ''targetZ'' est binaire pour un problème de classement. Nous optons naturellement pour un seul neurone en sortie (**Units = 1**) et une fonction d'activation sigmoïde (**Activation function = Sigmoid**). Il faut laisser le biais cette fois-ci.



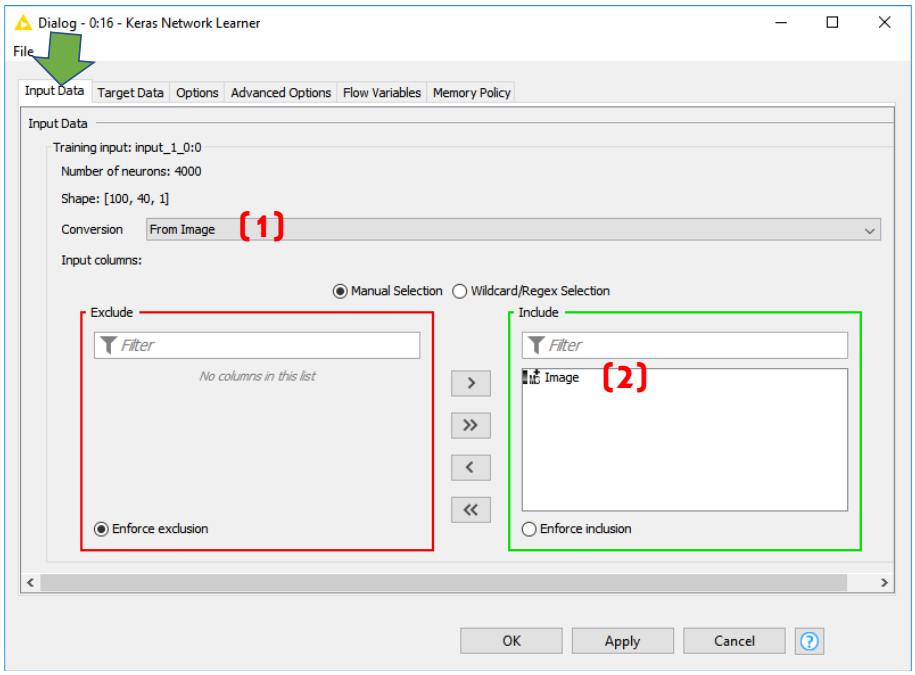

Ça y est, notre structure de réseau convolutif est prête.

## 4.2 Processus d'apprentissage

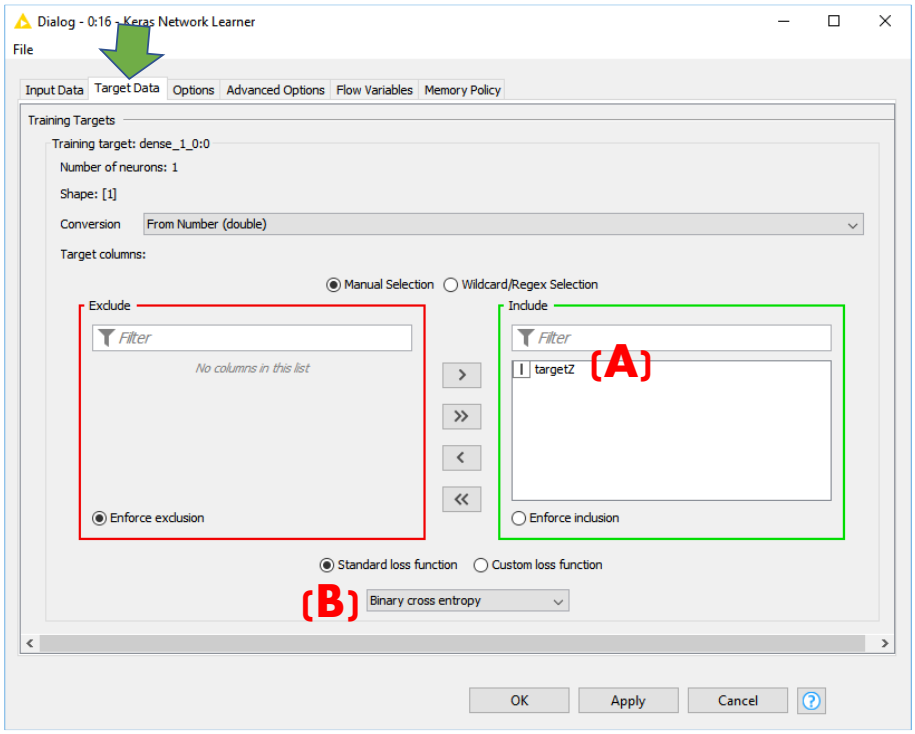
Pour le processus d'apprentissage, nous utilisons le composant **KERAS NETWORK LEARNER** (voir aussi ''Deep Learning avec Keras sous Knime'', juillet 2019 ; section 4.3.3). Nous lui connectons les données d'apprentissage issu de PARTITIONNING et la dernière couche du réseau (KERAS DENSE LAYER). Voici notre workflow à ce stade.



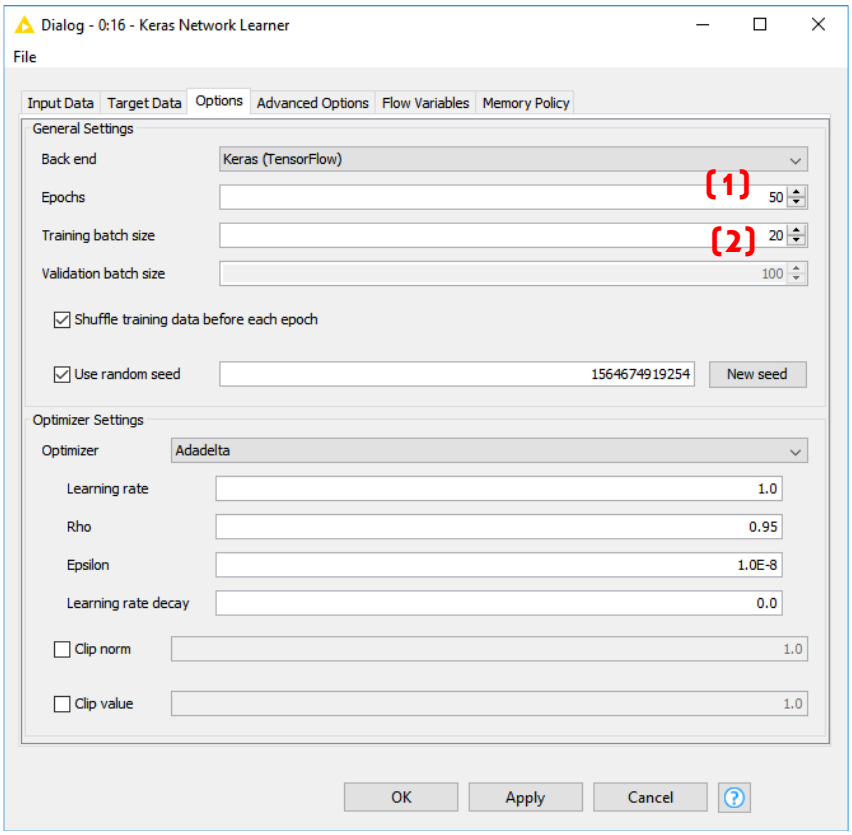

Dans la boîte de paramétrage, pour l'onglet INPUT DATA, nous précisons que le réseau prend les images en entrée **(1)** et nous précisons la variable concernée **(2)**.



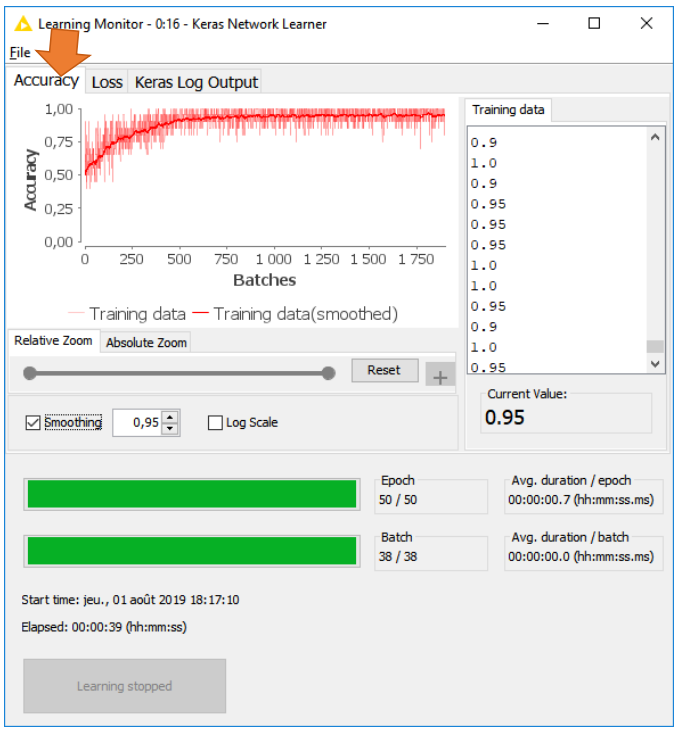

Dans TARGET DATA, nous précisons : la variable cible ''targetZ'' **(A)** et la fonction de perte standard ''Binary cross entropy'' **(B)**.

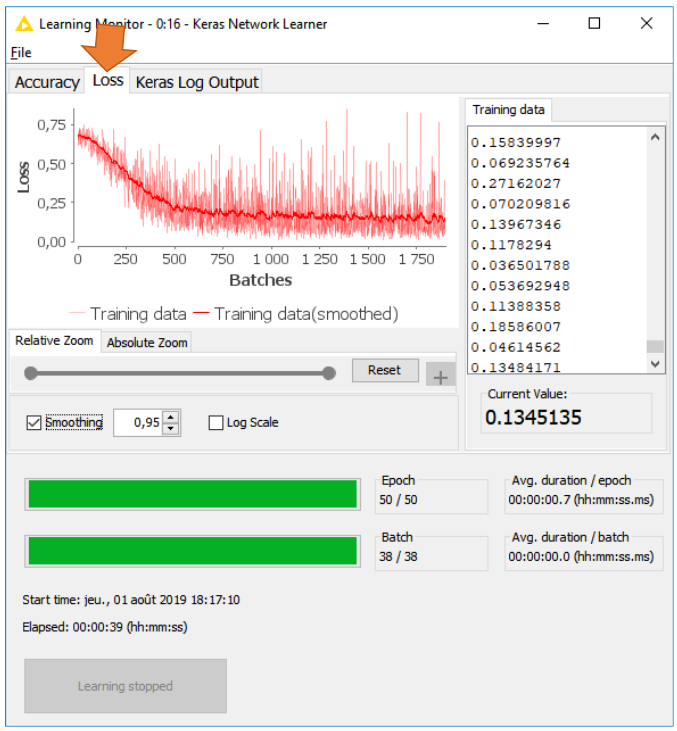



Dans OPTIONS, nous fixons le nombre de passage sur la base à 50 (**Epochs**) **(1)** et la taille des lots à 20 (**Training batch size**) **(2)** c.-à-d. les poids synaptiques sont recalculés tous les passages de 20 observations.



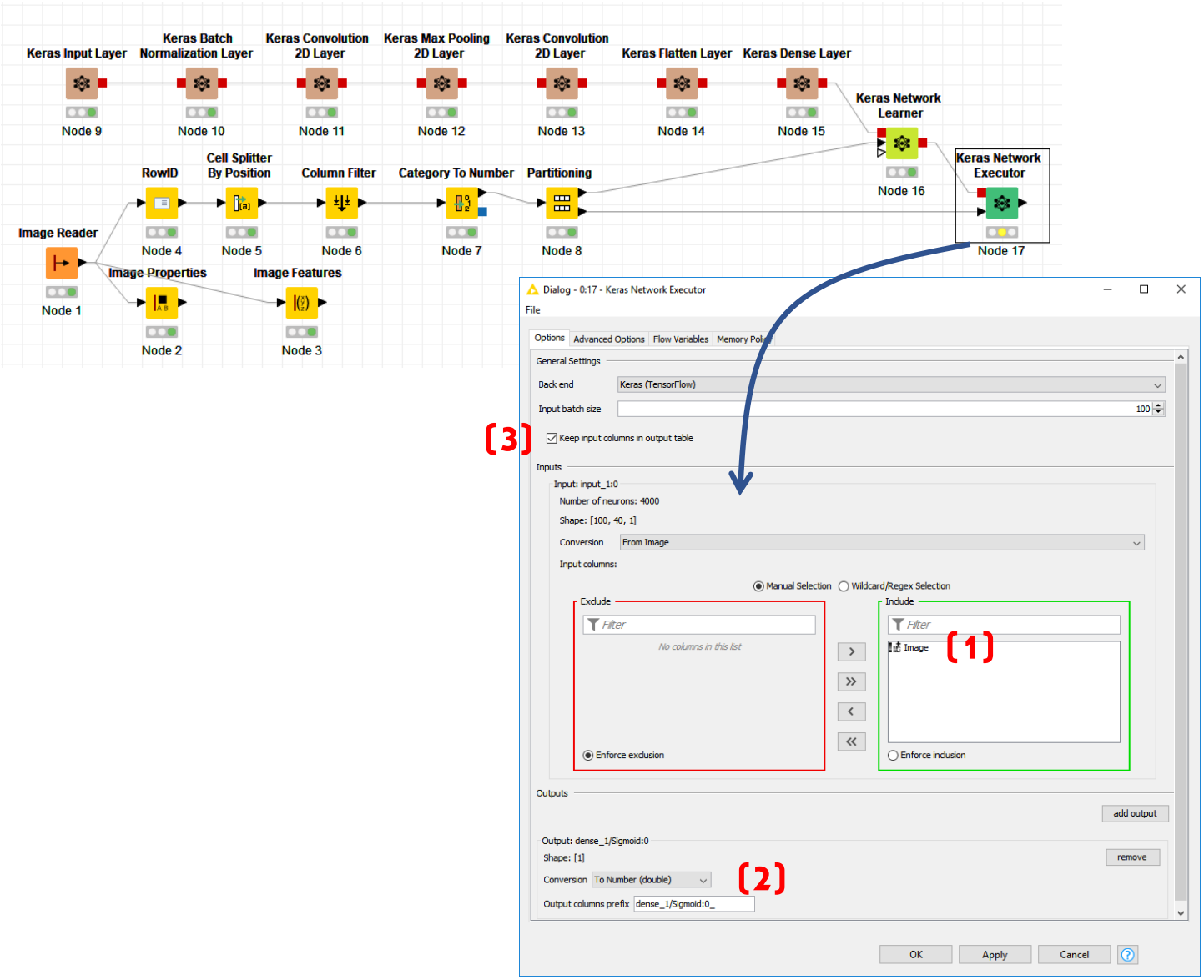

Nous actionnons le menu EXECUTE AND OPEN VIEWS pour lancer le processus d'apprentissage. Dans la fenêtre de suivi (LEARNING MONITOR) qui apparaît automatiquement, tant la courbe du taux de reconnaissance (**Accuracy**) que celle de la valeur de perte (**Loss**) laissent à penser que la modélisation est couronnée de succès.



## 4.3 Prédiction en test

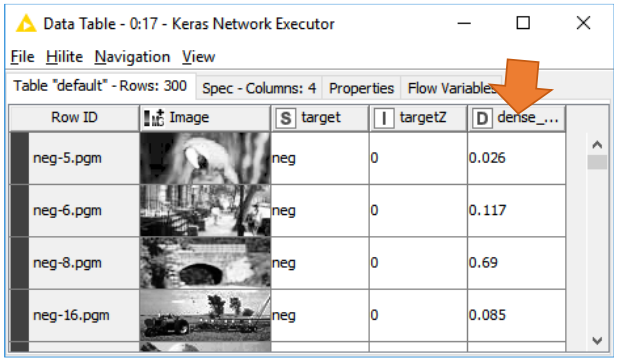
Mais pour s'en assurer de manière objective, rien ne vaut l'évaluation sur un échantillon test. Nous réalisons la prédiction sur les images que nous avions mis de côté précédemment (section 3.6) à l'aide du composant **KERAS NETWORK EXECUTOR** connecté au KERAS NETWORK LEARNER et à la seconde sortie du PARTITIONING.



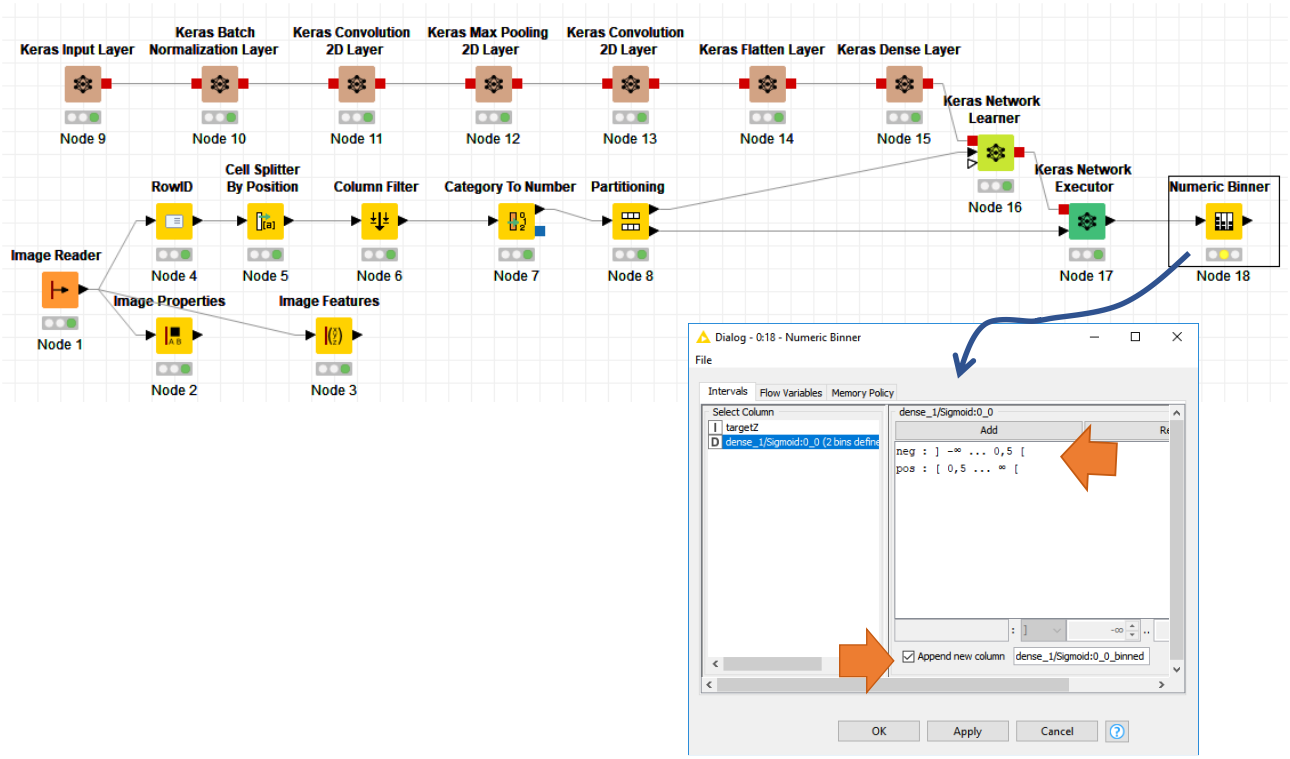

Il prend en entrée les images **(1)**, son output ''ADD OUTPUT'' correspond à la sortie de la couche dense du réseau **(2)**. Nous n'oublions de conserver les données initiales dans le dataset **(3)**. En effet, nous avons besoin de la variable ''target'' pour effectuer la confrontation en test entre classes observées et classes prédites.

Après exécution, nous pouvons visualiser l'ensemble de données courant en cliquant sur le menu DATA TABLE. Nous disposons des images, de la classe observée (target) recodée (targetZ), et

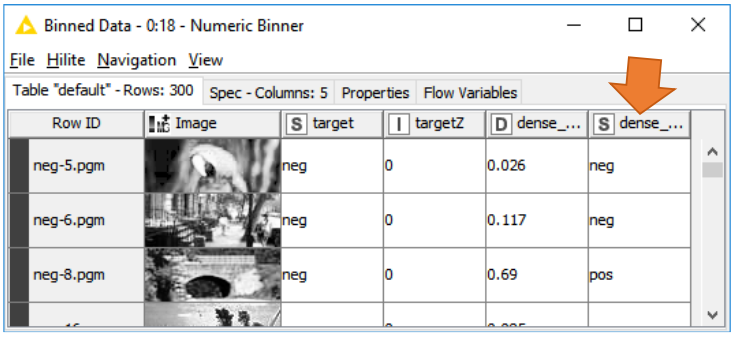
de la prédiction du réseau. Elle correspond à des valeurs comprises entre 0 et 1 puisque nous avons utilisé une fonction de transfert sigmoïde.




Il nous faut la convertir en prédiction (pos / neg) en la confrontant à la valeur seuil 0.5. Nous utilisons l'outil **NUMERIC BINNER** que nous paramétrons comme suit.

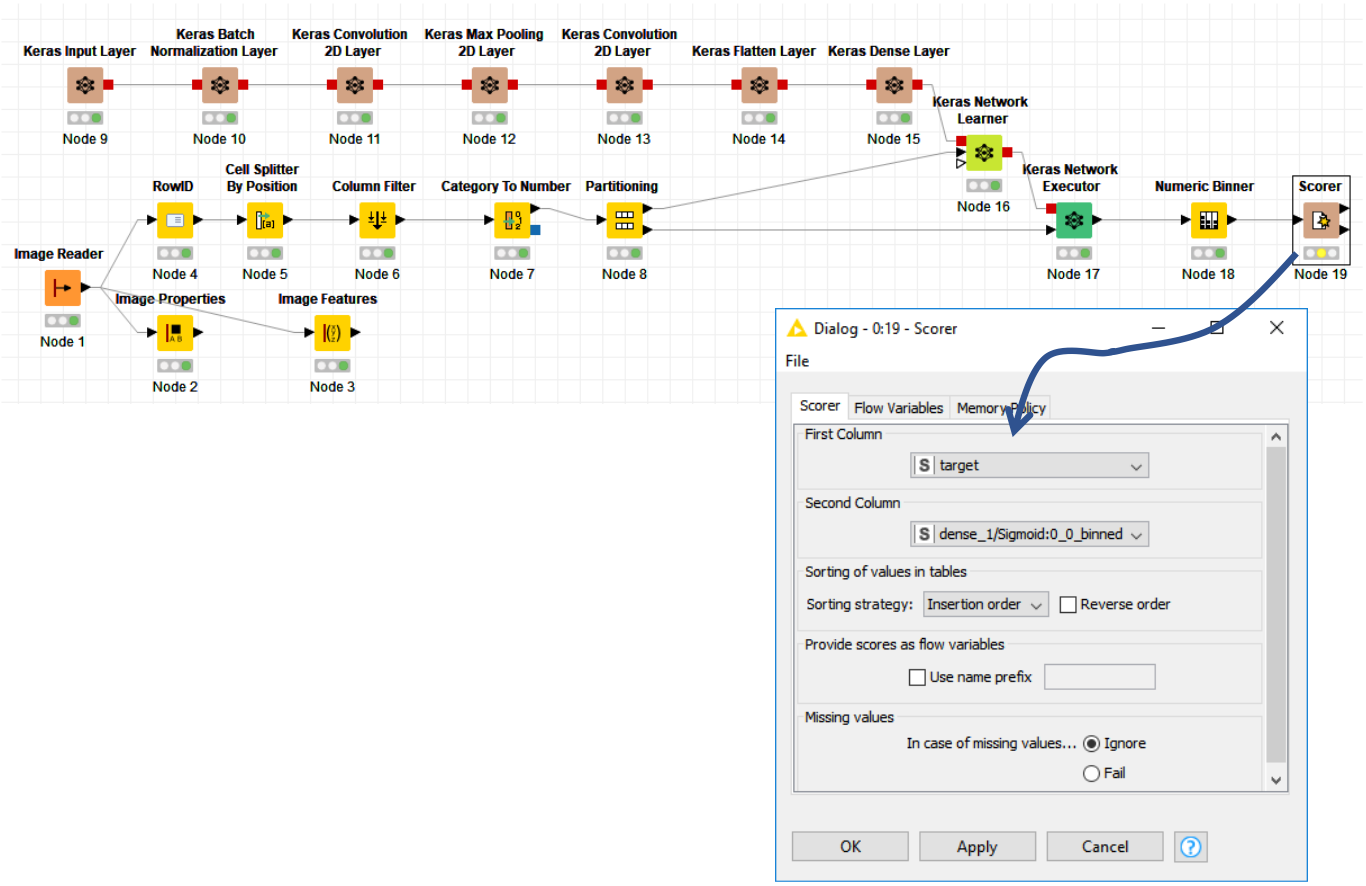



Après EXECUTE, la prédiction apparaît lorsque nous actionnons le menu BINNED DATA.

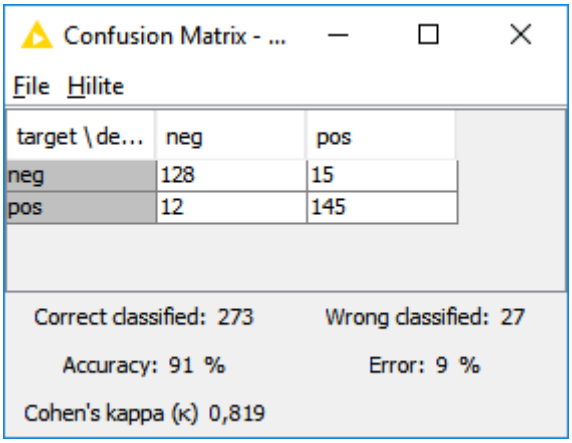


## 4.4 Matrice de confusion et taux d'erreur

Il ne reste plus qu'à confronter classes prédites et observées avec le composant **SCORER**. Nous le paramétrons de manière à opposer (en ligne) la variable TARGET et (en colonne) la prédiction DENSE_1/SIGMOID:0_0_BINNED.




Un dernier EXECUTE AND OPEN VIEWS pour la route…




… et KNIME nous annonce un taux de reconnaissance de 91%. Sur les 300 images de l'ensemble de test, 273 ont été reconnues correctement.

# 5 Conclusion

Bien sûr, je me suis volontairement contenté d'un réseau de taille limitée dans ce tutoriel. Il vous appartient de le sophistiquer en rajoutant des couches, en les combinant au mieux, en jouant sur les paramètres. Attention, le jeu devient très vite addictif. J'ai moi-même passé des heures à essayer telle ou telle solution, en faisant mouliner à max mon pauvre PC qui peinait sous la canicule. Les couinements plaintifs de ses ventilateurs internes de refroidissement m'ont dissuadé d'aller trop loin. Je suis finalement revenu à la raison en implémentant une solution simple que je peux décrire sans prendre le risque de perdre le lecteur.

Notons que KNIME a mis en ligne des exemples d'utilisation des CNN pour le traitement des bases MNIST et ''Cats and Dogs''. Les réseaux sont construits à l'aide de composants qui permettent d'exécuter du code Python. La portée pédagogique du didacticiel n'est pas vraiment avérée je trouve. Autant réaliser l'ensemble des traitements sous Python dans ce cas. La cohérence y gagne. Mais il me semblait important de les signaler. Charger les workflow et inspecter le code Python est très instructif, notamment concernant les stratégies à adopter pour empiler les couches des réseaux.

# 6 Références

Dumoulin V., Visin F., « A guide to convolution arithmetic for deep learning », mars 2016.

KNIME Deep Learning – Keras Integration -- https://www.knime.com/deeplearning/keras

Tufféry S., « Big Data, Machine Learning et Apprentissage profond », Technip, 2019.

Tutoriel Tanagra, « Deep Learning avec Keras sous Knime », juillet 2019.

Tutoriel Tanagra, « Image mining avec Knime », juin 2016.