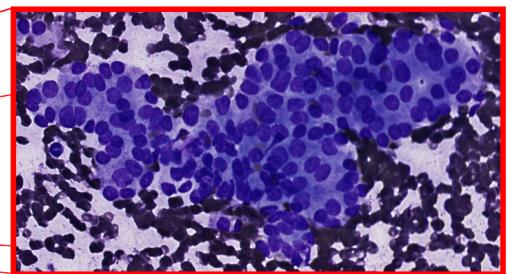
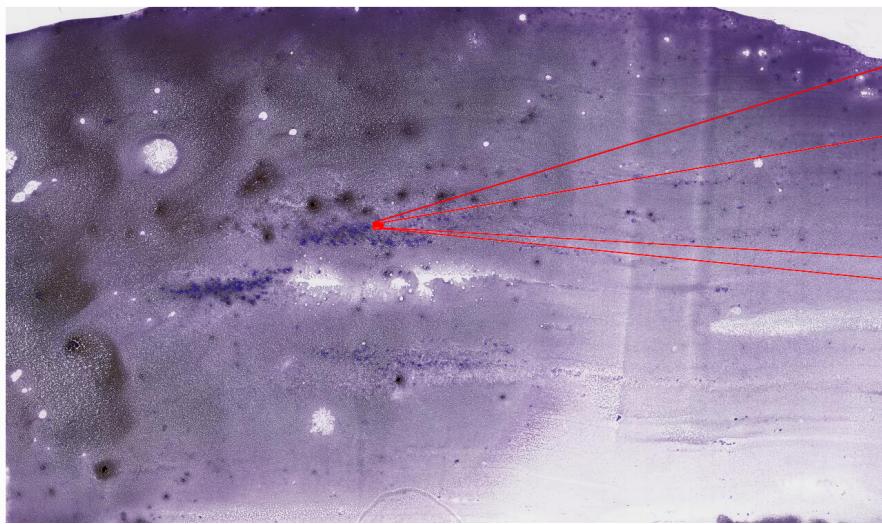


# **TensorFlow and Keras (deep learning by Google)**

Romain Mormont

# "I, Me, Mine"

- Doctorant en machine learning (ML) à Montefiore sous la supervision de Pierre Geurts et Raphaël Marée
- Recherche: machine learning appliqué au traitement de (très) grandes images médicales



↗ **x130**

# Au programme

- Machine et deep learning
- Frameworks de deep learning, TensorFlow et Keras
- Deep learning avec TensorFlow et Keras
  - Perceptron binaire
  - Perceptron multicouche
  - Réseaux convolutifs
  - Transfer learning

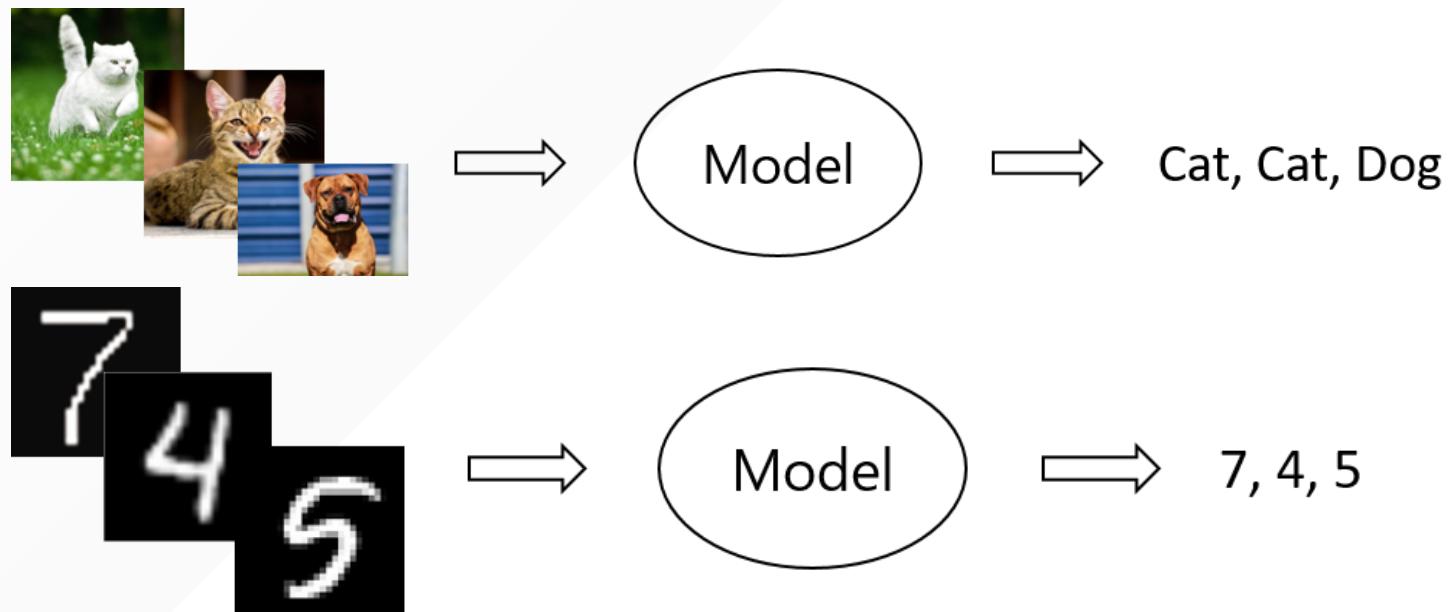
# Au programme

- Machine et deep learning
- Frameworks de deep learning, TensorFlow et Keras
- Deep learning avec TensorFlow et Keras
  - Perceptron binaire
  - Perceptron multicouche
  - Réseaux convolutifs
  - Transfer learning

# Machine learning ?

Le **machine learning** (supervisé) est un ensemble de méthodes permettant à un système informatique de construire/d'apprendre un modèle entrée(s)-sortie(s) sur base d'un ensemble de données.

Un **modèle** peut être vu comme une relation entre une ensemble d'entrées (i.e. les variables) et une sortie. Il possède des **paramètres** que l'on peut modifier afin l'adapter à un problème cible. Par exemple:

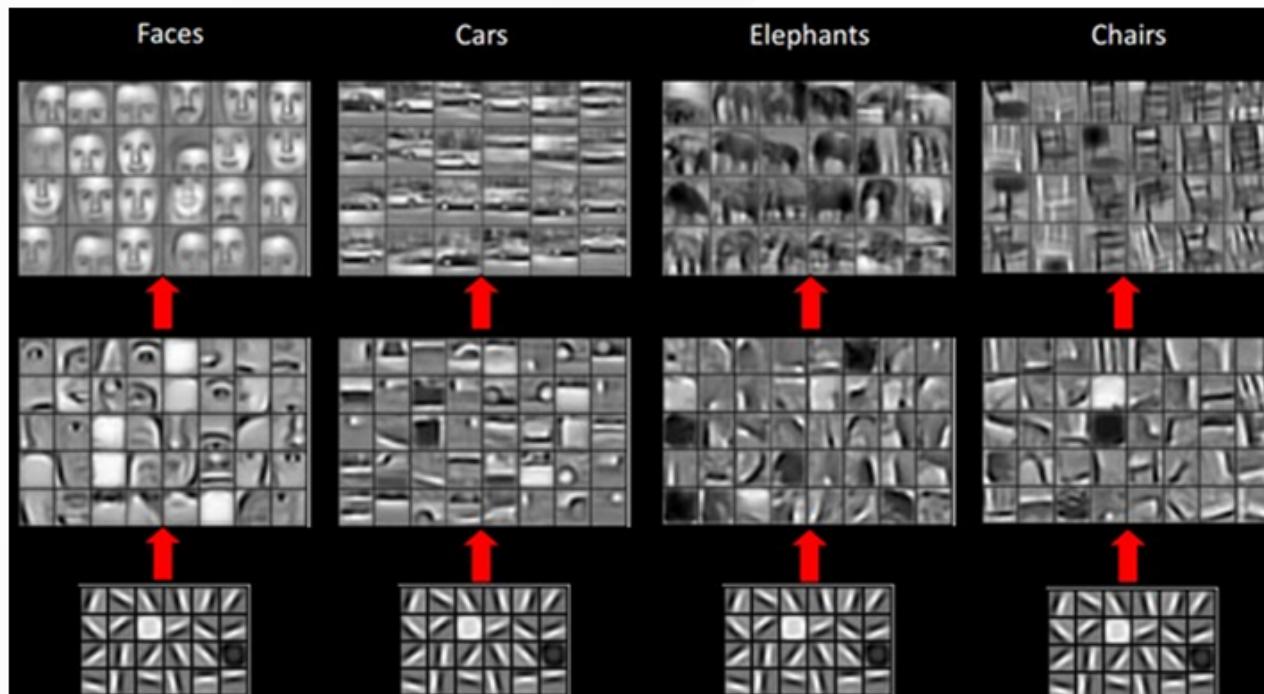


# Deep learning ?

Le **deep learning** est un ensemble de méthodes de machine learning basées sur l'apprentissage de **hierarchies de descripteurs** (*hierarchical features*).

Un **descripteur** (*feature*) est une information extraite d'une entité dans le but de la décrire.

En *vision par ordinateur*: pixel > bord > texton > motif > morceau > objet

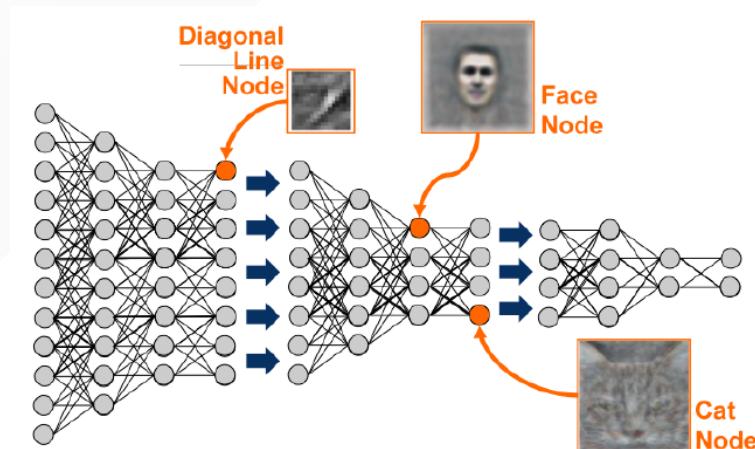


# Deep learning ?

Autres exemples:

- *reconnaissance vocale*: échantillon > bande spectrale > formant > motif > phonème > mot
- *traitement du langage naturel*: caractère > mot > groupe nominal/verbal > clause > phrase > histoire

Les méthodes de deep learning exploitent ces hiérarchies à l'aide de réseaux neuronaux en couche pour obtenir des modèles performants.



# Deep learning ?

Travailler avec un modèle sur une tâche cible implique en général deux grandes étapes:

- **entraînement** (*training, learning*): on optimise les paramètres du modèle afin d'améliorer ses performances. Procédure classique:
  1. on fournit au modèle un sous-ensemble aléatoire des données d'entraînement (*batch*) pour lesquels on connaît l'objectif et il retourne un ensemble de prédictions  $\hat{y}$
  2. on indique au modèle à quel point ses prédictions  $\hat{y}$  sont erronées à l'aide d'une **fonction de perte/d'erreur** (*loss function*)
  3. sur base de l'erreur, les paramètres du modèle sont ajustés pour améliorer ses performances (via **backpropagation**)
  4. on répète les étapes 1 à 3 jusqu'à ce que le modèle soit suffisamment performant
- **inférence** (*prediction, inference*): utilisation du modèle afin de produire une prédition sur des nouvelles données

# Au programme

- Machine et deep learning
- Frameworks de deep learning, TensorFlow et Keras
- Deep learning avec TensorFlow et Keras
  - Perceptron binaire
  - Perceptron multicouche
  - Réseaux convolutifs
  - Transfer learning

# Deep learning frameworks

Beaucoup de frameworks disponibles:

- **TensorFlow** (by Google)
- **Keras** (intégré dans TensorFlow depuis 01/2017)
- **Torch/PyTorch** (by Facebook)
- **Caffe2** (by Facebook)
- **CNTK** (by Microsoft)
- **MXNet** (sponsored by the Apache Incubator)
- **DL4j** (by Skymind)
- **Theano** (fin du support annoncée le 28/09/2017)
- ...

# TensorFlow

D'après Google: "*An open-source software library for Machine Intelligence*"

Quelques dates:

- 09/11/15: [open-source release](#)
- 03/01/17: [choix de Keras](#) comme interface haut-niveau
- 15/02/17: [version 1.0.0](#)
- 06/11/17: [dernière release stable \(1.4.0\)](#)

Quelques chiffres (sur GitHub):

- 1134 contributeurs
- 24117 commits, ~250 par semaine
- 76199 followers
- 1162 issues ouvertes (7689 fermées)

# TensorFlow

- Librairie de mathématique symbolique
- Interface de programmation en Python mais cœur en C++
- Graphe de calcul statique: les noeuds sont les opérations et les arêtes sont les tenseurs
  - Le graphe doit être compilé avant d'être utilisé
  - Optimisations possibles à la compilation
  - ! Ne permet pas d'implémenter des comportements dynamiques
- Support deep learning: beaucoup de composants de base disponibles
- Programmation déclarative: le graphe de calcul est construit de manière déclarative
  - Un pas vers l'impératif avec Eager (pre-alpha)
- Support (multi) GPU
- TensorBoard: outil de visualisation, monitoring temps réel via une interface web

# TensorFlow

## Graphe de calcul: définition et exécution

Implémentons l'opération suivante en TensorFlow:

$$y = \sqrt{a + \sin(b)}$$

```
from math import pi
import tensorflow as tf

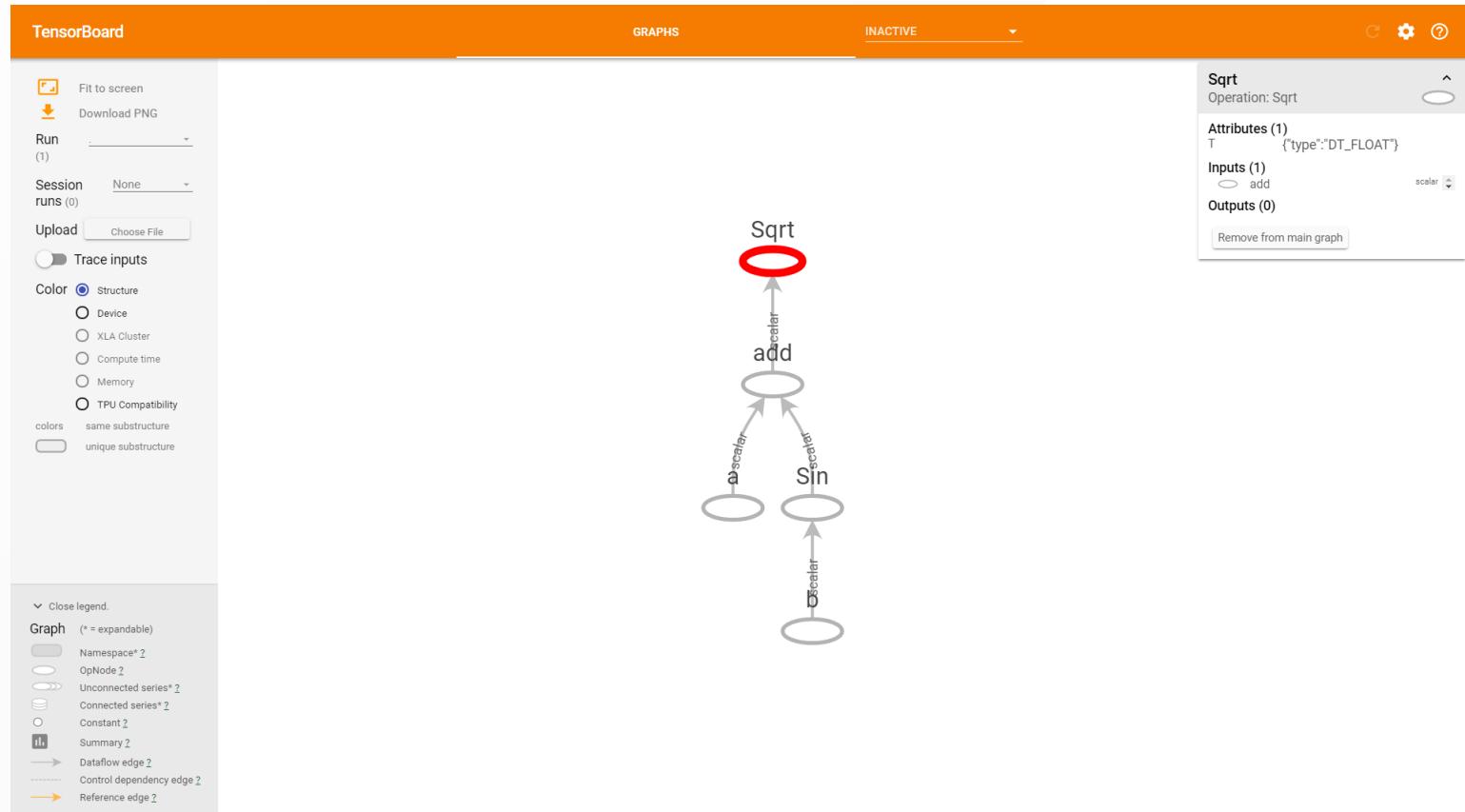
if __name__ == "__main__":
    # graph definition
    a = tf.placeholder(tf.float32, shape=(), name="a")
    b = tf.placeholder(tf.float32, shape=(), name="b")
    radicand = a + tf.sin(b)
    y = tf.sqrt(radicand)

    # execution
    with tf.Session() as sess:
        feed = {a: 3, b: pi / 2.0}
        res, = sess.run([y], feed_dict=feed)

        print("y: {}".format(res))
```

# TensorFlow

## Graphe de calcul: visualisation avec TensorBoard



# Keras

Keras, d'après son créateur:

*Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.*

Philosophie:

- **User-friendliness**: développé de manière à réduire la charge cognitive du développeur (i.e. API cohérente et simple, feedback d'erreur clair et exploitable, use cases communs implémentables en un nombre réduit d'étapes...)
- **Modularité**: la librairie est structurée comme un ensemble de modules indépendants qui peuvent être combinés pour créer des modèles
- **Extensibilité**: de nouveaux modules peuvent être créés et intégrés de manière simple

# Au programme

- Machine et deep learning
- Frameworks de deep learning, TensorFlow et Keras
- Deep learning avec TensorFlow et Keras
  - Perceptron binaire
  - Perceptron multicouche
  - Réseaux convolutifs
  - Transfer learning

# Deep learning with TensorFlow

Concentrons-nous sur un problème en particulier: la reconnaissance d'image et plus particulièrement la **reconnaissance de chiffres écrits à la main**.

- Chaque image contient un chiffre de 0 à 9
- Image en noir et blanc, taille 28x28 pixels
- **Objectif:** étant donnée l'image, prédire le chiffre qu'elle contient

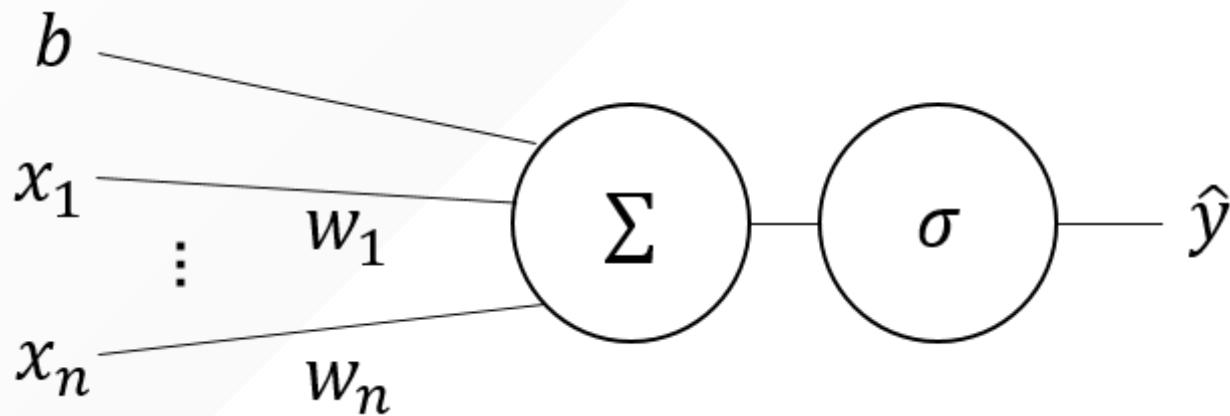
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 2 | 1 | 9 | 5 | 6 | 2 | 1 | 8 | 6 | 1 | 9 | 3 | 9 | 6 | 3 | 9 | 6 | 5 |
| 8 | 9 | 1 | 2 | 5 | 0 | 0 | 6 | 6 | 4 | 6 | 1 | 3 | 0 | 4 | 3 | 6 | 1 | 3 | 2 |
| 6 | 7 | 0 | 1 | 6 | 3 | 6 | 3 | 7 | 0 | 6 | 3 | 9 | 5 | 1 | 5 | 3 | 9 | 6 | 5 |
| 3 | 7 | 7 | 9 | 4 | 6 | 6 | 1 | 8 | 2 | 6 | 1 | 9 | 3 | 9 | 6 | 3 | 6 | 1 | 3 |
| 2 | 9 | 3 | 4 | 3 | 9 | 8 | 7 | 2 | 5 | 6 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 |
| 1 | 5 | 9 | 8 | 3 | 6 | 5 | 7 | 2 | 3 | 6 | 1 | 3 | 2 | 5 | 3 | 2 | 5 | 3 | 2 |
| 9 | 3 | 1 | 9 | 1 | 5 | 8 | 0 | 8 | 4 | 9 | 3 | 1 | 9 | 1 | 5 | 3 | 2 | 5 | 3 |
| 5 | 6 | 2 | 6 | 8 | 5 | 8 | 8 | 9 | 9 | 6 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 2 | 1 |
| 3 | 7 | 7 | 0 | 9 | 4 | 8 | 5 | 4 | 3 | 6 | 1 | 3 | 2 | 5 | 3 | 2 | 5 | 3 | 2 |
| 7 | 9 | 6 | 4 | 7 | 0 | 6 | 9 | 2 | 3 | 6 | 1 | 3 | 0 | 6 | 3 | 0 | 6 | 1 | 3 |

# (Deep) learning with TensorFlow

## Perceptron binaire > modèle (i)

Commençons par essayer de distinguer deux chiffres  $c_0$  et  $c_1$  avec un modèle simple (et superficiel): le **perceptron binaire**.

$$\hat{y} = \sigma \left( \sum_{i=1}^N w_i x_i + b \right)$$

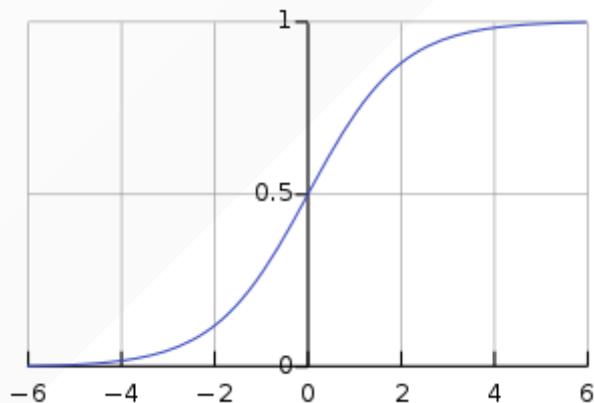


On optimise **les paramètres** du perceptron (i.e. les poids  $w_i$  et le biais  $b$ ) afin que le modèle puisse distinguer les deux chiffres.

# (Deep) learning with TensorFlow

## Perceptron binaire > modèle (ii)

- le **perceptron est aussi appelé neurone** car inspiré du fonctionnement de la cellule cérébrale
- l'opérateur  $\sigma(\cdot)$  est la fonction sigmoïde:  $\sigma(x) = \frac{1}{1+e^{-x}}$



- $\sigma(\cdot)$  est la **fonction d'activation** du neurone
- $\hat{y}$  est la probabilité que le chiffre soit  $c_1$
- pour déterminer le chiffre, on choisit  $c_0$  si  $\hat{y} \leq 0.5$ ,  $c_1$  sinon

# (Deep) learning with TensorFlow

## Perceptron binaire > modèle (iii)

Pour exploiter les capacités de TensorFlow, on va transformer le perceptron en un **problème vectoriel**.

$$\hat{y} = \sigma(\mathbf{x}^T \mathbf{w} + b)$$

- **entrée**: un vecteur  $\mathbf{x}$  de 784 éléments
- **poids**: un vecteur  $\mathbf{w}$  de 784 éléments
- **biais**: un scalaire  $b$

En pratique, on veut que le modèle puisse **traiter plusieurs images à la fois** !

Dans le code, on va donc plutôt utiliser une matrice d'entrée  $\mathbf{X}$  de taille ( $B \times 784$ ) où  $B$  est le nombre d'images.

# (Deep) learning with TensorFlow

## Perceptron binaire > modèle (iv)

Entrées et variables:

- entrée du modèle  $\mathbf{X} \rightarrow \text{tf.placeholder}$

```
x = tf.placeholder(shape=[None, 784], dtype=tf.float32, name='x')
```

- variables du modèle  $\mathbf{w}$  et  $b \rightarrow \text{tf.Variable}$

```
w = tf.Variable( # weights
    initial_value=tf.truncated_normal(shape=[784, 1]),
    trainable=True, name="w"
)
```

```
b = tf.Variable( # bias
    initial_value=tf.zeros(shape=[1], dtype=tf.float32),
    trainable=True, name="b"
)
```

# (Deep) learning with TensorFlow

## Perceptron binaire > modèle (v)

Opérations:

- produit matriciel → `tf.matmul`

```
mult = tf.matmul(x, w)
```

- ajouter le biais → `tf.nn.bias_add`

```
with_bias = tf.nn.bias_add(mult, b)
```

- sigmoïde → `tf.nn.sigmoid`

```
out = tf.nn.sigmoid(with_bias, name="out")
```

Notons l'utilisation de `tf.nn`, le **module *neural networks* de TensorFlow**.

# (Deep) learning with TensorFlow

## Perceptron binaire > entraînement > loss

Pour optimiser le modèle, il faut pouvoir quantifier son erreur avec la fonction de perte.

Entropie croisée binaire (*binary cross-entropy*):

$$\mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

- $y$  identifie le véritable chiffre pour une image d'entrée. Il vaut 0 si le chiffre est  $c_0$ , 1 sinon.

```
y = tf.placeholder(shape=[None, 1], dtype=tf.float32, name="y")
```

- implémentation de la fonction de perte

```
ce = - y * tf.log(out) - (1 - y) * tf.log(1 - out)
loss = tf.reduce_mean(ce, name="loss")
```

# (Deep) learning with TensorFlow

## Perceptron binaire > entraînement > optimisation

Il faut maintenant définir la **stratégie d'optimisation** que TensorFlow va utiliser pour mettre à jour le modèle.

Nous allons utiliser une **descente de gradient**:

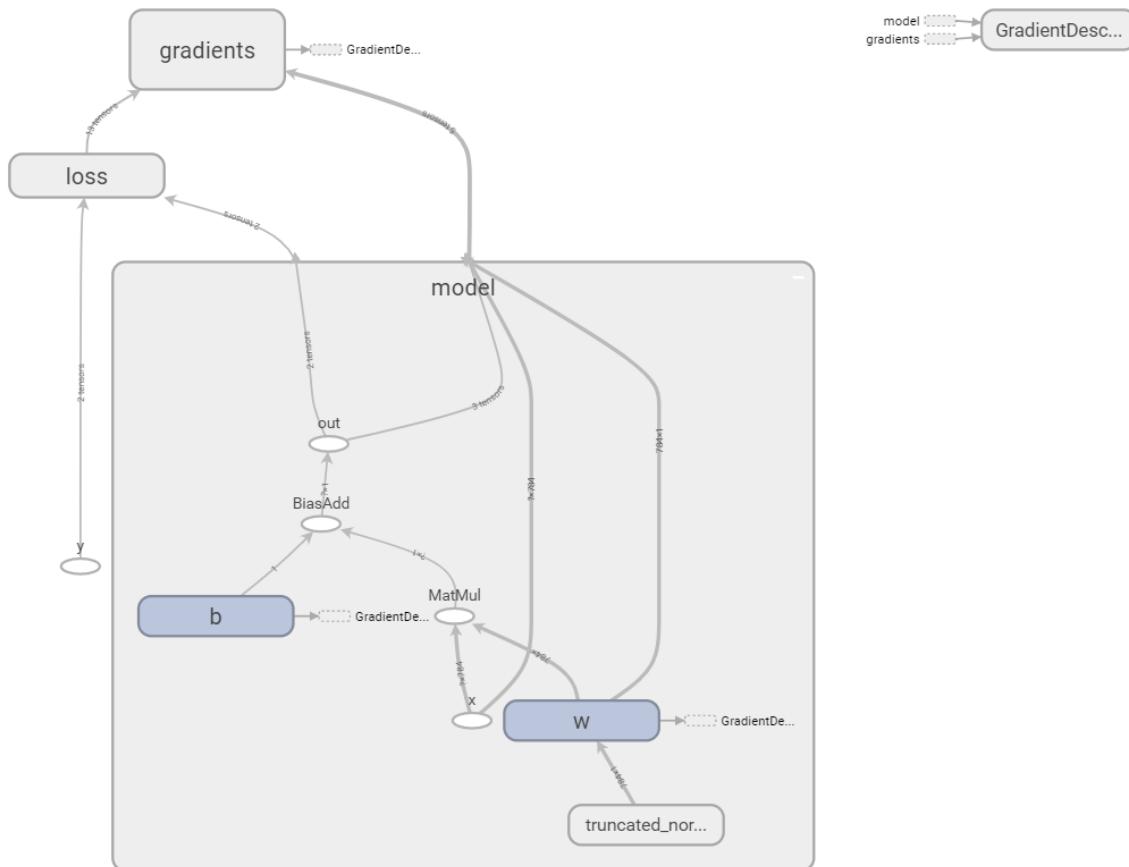
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=1e-3)
minimize = optimizer.minimize(loss)
```

Beaucoup d'**autres méthodes** sont implémentées dans TensorFlow:

- AdamOptimizer
- MomentumOptimizer
- RMSPropOptimizer
- ...

# (Deep) learning with TensorFlow

## Perceptron binaire



# (Deep) learning with TensorFlow

## Perceptron binaire > entraînement > code

```
initializer = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run([initializer])

    for i in range(1000):
        feed = {
            x: # random batch of images...
            y_true: # and their classes
        }
        _loss, _ = sess.run([loss, minimize], feed_dict=feed)
        print("At iter {}, loss is {}".format(i, _loss))
```

- On exécute l'opération `minimize` un certain nombre d'itérations
- À chaque itération, on fournit un batch aléatoire
- Et .. c'est tout !

TensorFlow fait beaucoup de choses pour nous dans l'ombre !

# Deep learning with TensorFlow

## Vers un modèle plus réaliste...

Le **perceptron binaire** n'est jamais utilisé seul dans les applications réelles car  
**ce modèle est trop simple.**

- les **problèmes réels** sont **trop complexes** pour le perceptron
- il est limité à des **problèmes binaires**
- il est **superficiel** et ne permet pas d'exploiter la **nature hiérarchique** de certains problèmes

Nous allons étudier un **autre modèle** qui:

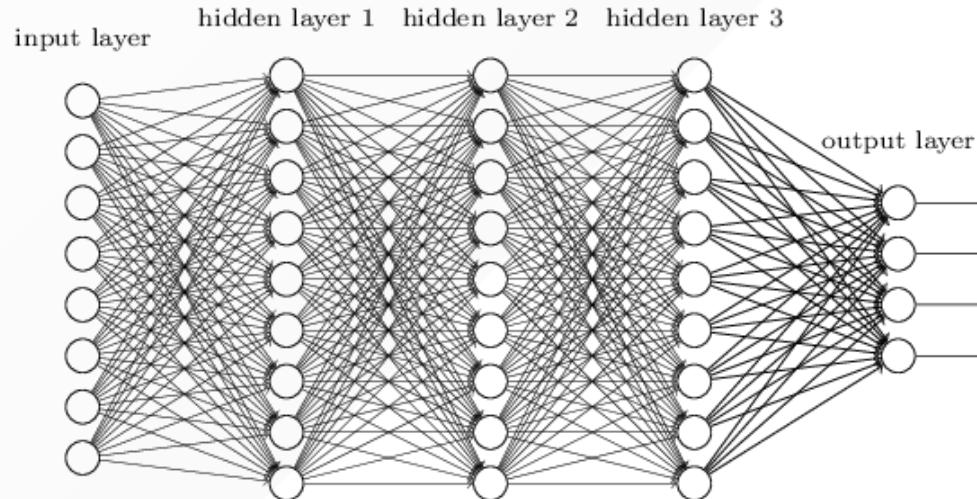
- combine des perceptrons
- tente de résoudre les problèmes évoqués ci-dessus

**Ce modèle est le perceptron multicouche**  
*(multi-layer perceptron)*

# Deep learning with TensorFlow

## Perceptron multicouche

Ce modèle est un ensemble de perceptrons arrangés **en couches**.



- les couches internes s'appellent **couches cachées** (*hidden layer*)
- chaque neurone est connecté à tous les neurones de la couche précédente
- il y a autant de neurones que de classes dans la dernière couche

# Deep learning with TensorFlow

## Perceptron multicouche > définition (i)

- $L$  est le **nombre de couches**. La couche 0 est le vecteur d'entrée  $\mathbf{x}$
- $n_l$  est le **nombre de neurones** à la couche  $l \in [0, L]$
- $\mathbf{W}_l$  est la **matrice des poids** de la couche  $l \in [1, L]$ , (dim.  $n_{l-1} \times n_l$ )
- $\mathbf{b}_l$  est le **vecteur de biais** pour la couche  $l \in [1, L]$  (dim.  $n_l$ )
- **couche d'entrée (input layer,  $l = 0, n_l = 784$ ):**  $\mathbf{a}_0 = \mathbf{x}$
- **couche cachée (hidden layer,  $l \in [1, L - 1]$ ):**

$$\mathbf{a}_l = \sigma \left( \mathbf{a}_{l-1}^T \mathbf{W}_l + \mathbf{b}_l \right)$$

- **couche de sortie (output layer,  $l = L, n_l = 10$ ):**

$$\hat{\mathbf{y}} = \text{softmax} \left( \mathbf{a}_{L-1}^T \mathbf{W}_L + \mathbf{b}_L \right)$$

# Deep learning with TensorFlow

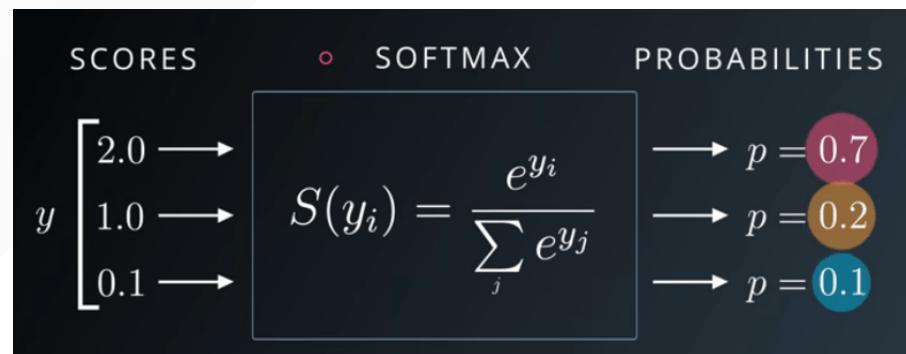
## Perceptron multicouche > définition (ii)

La fonction **softmax** est une nouvelle **fonction d'activation**:

- elle est définie comme suit: elle prend en entrée un vecteur  $\mathbf{x}$  de taille  $n$

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{k=1}^n e^{x_k}}$$

- $\text{softmax}(\mathbf{x})_i$  peut être interprété comme une probabilité
- l'élément  $\hat{y}_i$  de  $\hat{\mathbf{y}}$  **est la probabilité que le chiffre soit  $i$**



# Deep learning with TensorFlow

## Perceptron multicouche > implémentation (i)

- fonction générique pour créer un couche dense (*fully connected or dense layer*):

```
def layer(in_layer, in_size, out_size, activ="sigmoid", name=""):
    w = tf.Variable(
        initial_value=tf.truncated_normal([in_size, out_size]),
        trainable=True, name="{}/weights".format(name))
    bias = tf.Variable(
        initial_value=tf.zeros([out_size]),
        trainable=True, name="{}/bias".format(name))
    # compute activation
    prod = tf.matmul(in_layer, w)
    with_bias = tf.nn.bias_add(prod, bias)
    if activ == "softmax":
        return tf.nn.softmax(with_bias, name="{}/out".format(name))
    else:
        return tf.nn.sigmoid(with_bias, name="{}/out".format(name))
```

# Deep learning with TensorFlow

## Perceptron multicouche > implémentation (ii)

- vecteur de sortie  $\mathbf{y}$  ( $y_i$  vaut 1 si le véritable chiffre est  $i$ , 0 sinon):

```
y = tf.placeholder([None, 10], dtype=tf.float32, name="y")
```

- construction des couches cachées (ici 3 couches de tailles 64, 32 et 16):

```
prev_layer = x
prev_size = 784
for i, size in enumerate([64, 32, 16]):
    prev_layer = layer(
        in_layer=prev_layer,
        in_size=prev_size,
        out_size=size,
        activ="sigmoid",
        name="hidden_{}".format(i + 1)
    )
    prev_size = size
```

# Deep learning with TensorFlow

## Perceptron multicouche > implémentation (iii)

- la **couche de sortie**:

```
out = layer(
    in_layer=prev_layer,
    in_size=prev_size,
    out_size=10
    activ="softmax",
    name="output_layer"
)
```

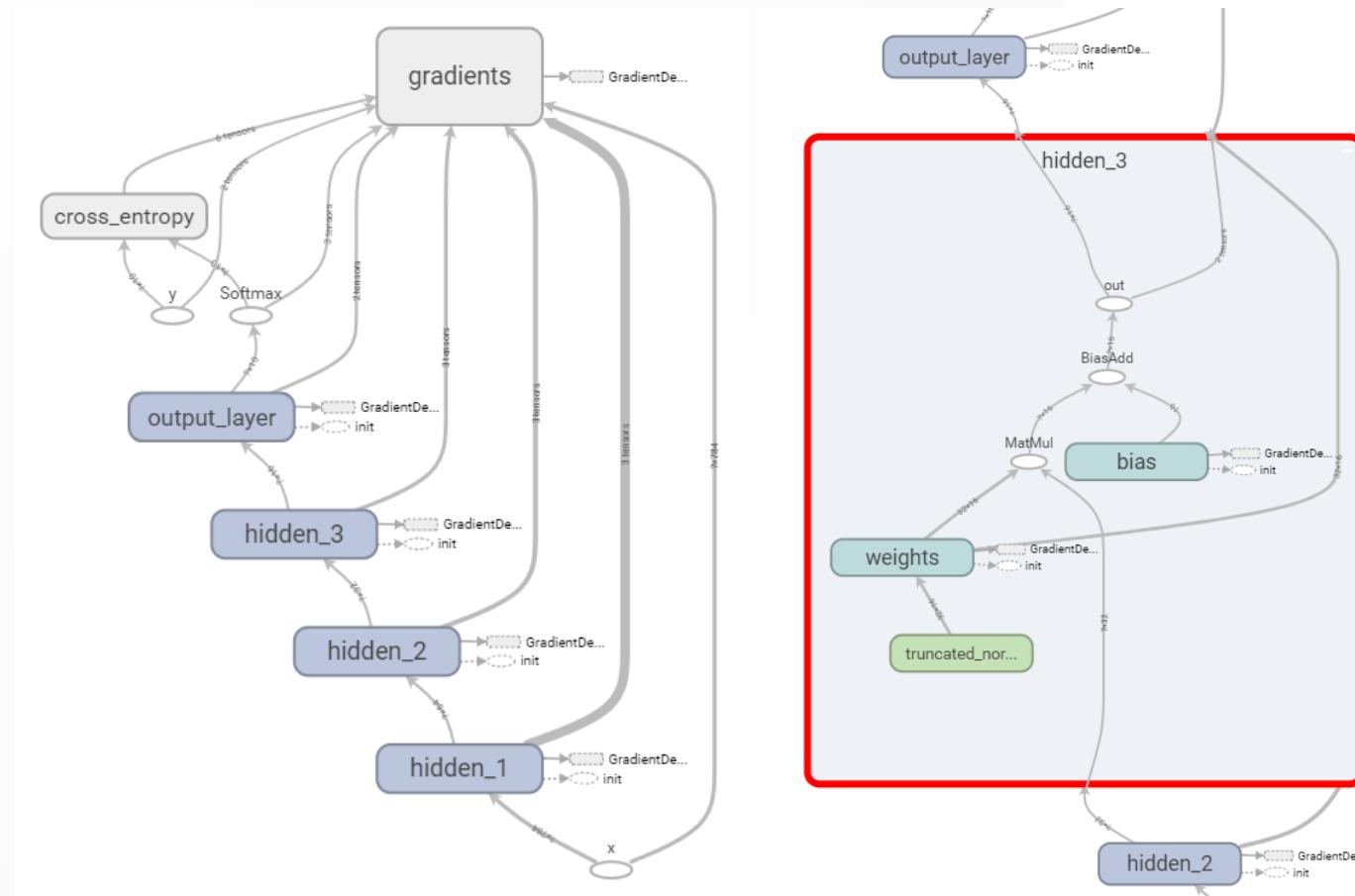
- la **fonction de perte**:  $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{10} y_i \log \hat{y}_i$

```
ce = - tf.reduce_sum(y * tf.log(out), axis=-1)
loss = tf.reduce_mean(ce)
```

- on utilise le **même code d'entraînement** !

# Deep learning with TensorFlow

Perceptron multicouche > graphe



# Deep learning with Keras

## Perceptron multicouche > implémentation (i)

Keras propose une interface de plus haut niveau que TensorFlow:

- le modèle est un objet de type Model
- un Model est une succession de Layer (par ex.: Dense pour une couche *fully connected*)

Pour construire le même réseau que précédemment:

```
input = Input(shape=(784,))

# create hidden layers
x = input
for size in [64, 32, 16]:
    x = Dense(size, activation="sigmoid", use_bias=True)(x)

# output layer
x = Dense(10, activation="softmax")(x)
model = Model(inputs=[input], outputs=[x])
```

# Deep learning with Keras

## Perceptron multicouche > implémentation (ii)

- c'est à la **compilation du modèle** qu'on précise la **fonction de perte** et la **stratégie d'optimisation**:

```
model.compile(  
    optimizer=sgd(lr=5e-2),  
    loss="categorical_crossentropy",  
    metrics=["accuracy"]  
)
```

- **entraînement** via la méthode `fit` de `Model`:

```
# train  
model.fit(  
    x=# all training images ...  
    y=# all training classes ...  
    batch_size=batch_size,  
    epochs=epochs  
)
```

# Deep learning with Keras

## Perceptron multicouche > implémentation (iii)

- fit écrit des informations sur la progression de l'entraînement sur la sortie standard:

```
Epoch 1/200  
55000/55000 [=====] - 4s - val_loss: 2.3002 - val_acc: 0.1  
Epoch 2/200  
55000/55000 [=====] - 1s - val_loss: 2.2999 - val_acc: 0.1  
Epoch 3/200%  
55000/55000 [=====] - 1s - val_loss: 2.2988 - val_acc: 0.1  
Epoch 4/200  
43648/55000 [=====>..] - ETA: 0s - loss: 2.2988 - acc: 0.1
```

- inférence via la méthode predict de Model:

```
y_pred = model.predict(new_images, batch_size=batch_size)
```

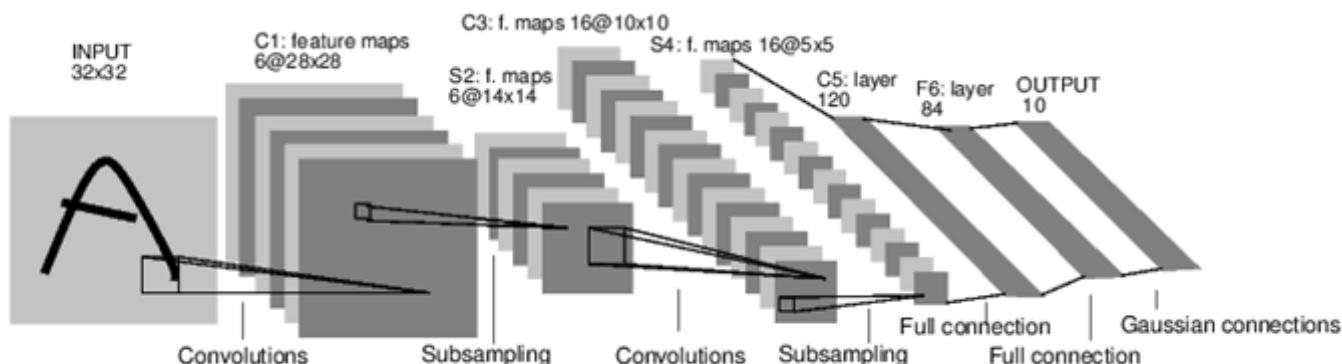
# Deep learning with Keras

Vers un modèle (encore) plus réaliste...

On obtient **97% d'exactitude** (*accuracy*) avec le perceptron multicouche...  
mais on peut faire mieux !

En pratique, les problèmes impliquants des images sont abordés avec des...

## Réseaux de neurones convolutifs (*convolutional neural networks*)



A Full Convolutional Neural Network (LeNet)

# Deep learning with Keras

## Réseaux convolutifs > structure

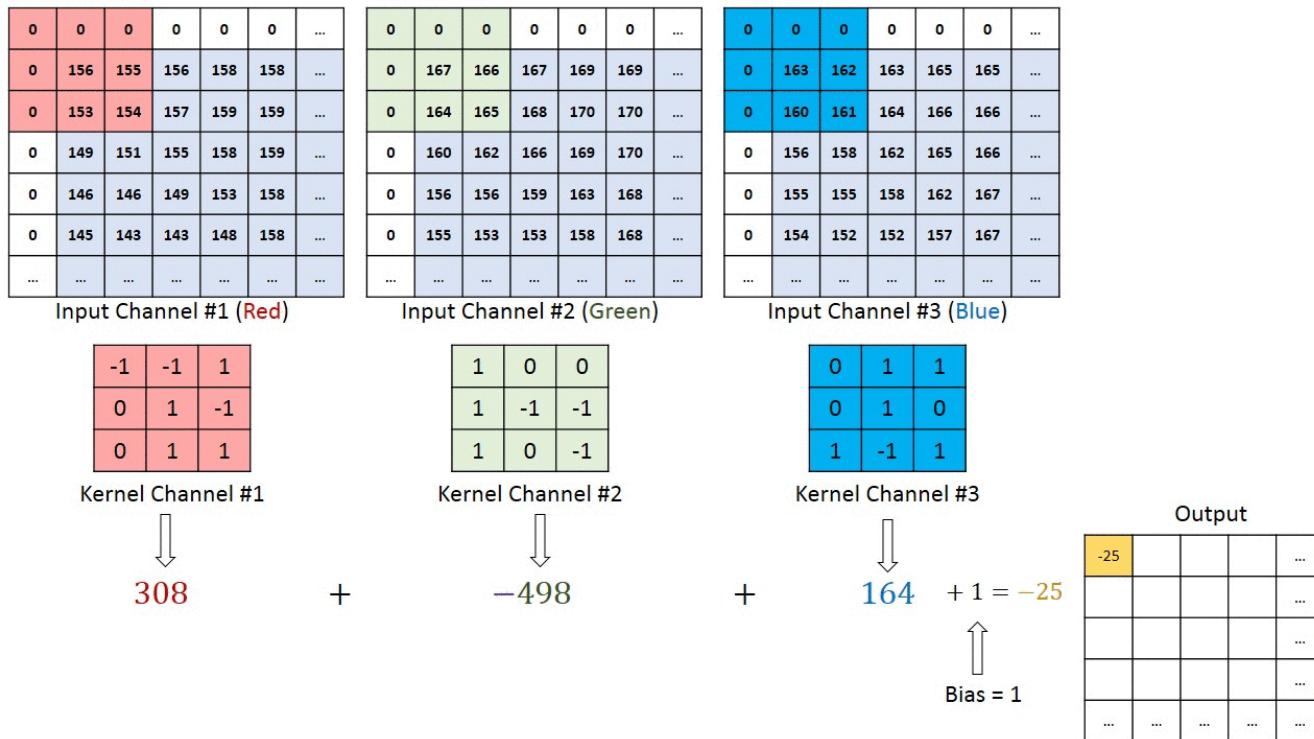
Un réseau convolutif est en général construit de la manière suivante:

- une série de **blocs de convolution** transformant leur entrée en **feature maps** avec les opérations suivantes:
  - une couche de **convolution**
  - une **fonction d'activation**
  - (*optionnel*) d'une couche de **pooling**
- un **perceptron multicouche**

# Deep learning with Keras

## Réseaux convolutifs > convolution (i)

Une **convolution** (à deux dimensions) est une **transformation linéaire** d'un **signal** (e.g. une image).



# Deep learning with Keras

## Réseaux convolutifs > convolution (ii)

En pratique:

- plusieurs filtres par couche
- les **noyaux sont appris !**

Paramètres de dimensionnement d'une couche de convolution:

- le **nombre de filtre**
- la **taille des noyaux** (*kernel size*)
- le **pas** des noyaux (*strides*)
- le **stratégie de gestion des bords** (*padding*)

Avec **Keras**:

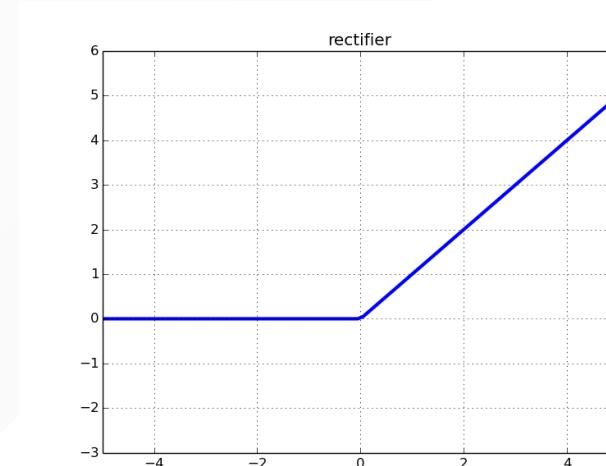
```
# arbitrary values: 32 3x3 kernels, with stride 1x1
x = Conv2D(filters=32, kernel_size=(3, 3), strides=1, padding="sam
```

# Deep learning with Keras

## Réseaux convolutifs > activation

On va utiliser une nouvelle fonction d'activation: l'**unité de rectification linéaire** (*rectified linear unit*, or *ReLU*):

$$\text{relu}(x) = \max(0, x)$$

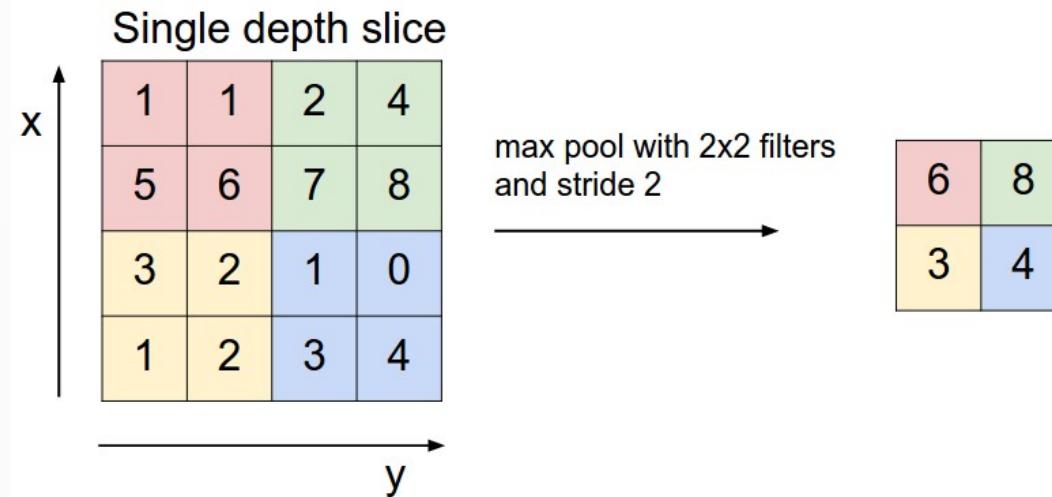


La fonction sera appliquée après chaque couche de convolution.

# Deep learning with Keras

## Réseaux convolutifs > pooling (i)

La **couche de pooling** sous-échantillonne son signal d'entrée. Plusieurs stratégies sont possibles: **max pooling, average pooling,...**



Le pooling apporte l'**invariance en translation**: le réseau peut détecter un objet quelque soit sa position dans l'image d'entrée.

# Deep learning with Keras

## Réseaux convolutifs > pooling (ii)

Paramètres de dimensionnement d'une couche de pooling:

- la **stratégie de pooling**
- la **taille des noyaux** (*pool size*)
- le **pas des noyaux** (*strides*)
- le **stratégie de gestion des bords** (*padding*)

En Keras:

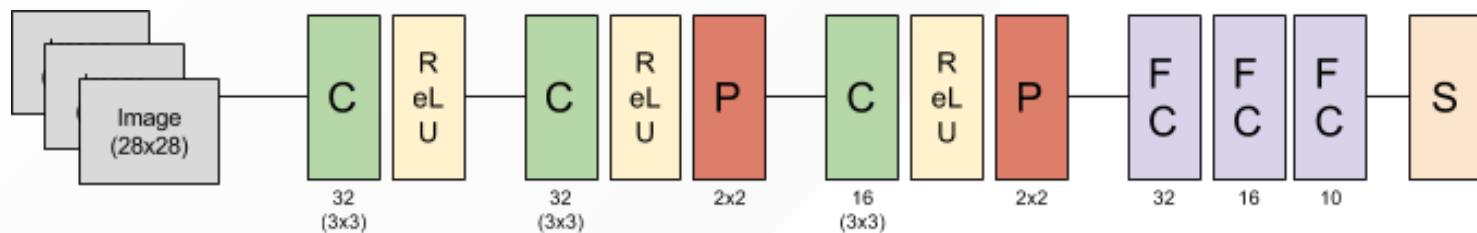
```
# max pooling with a 2x2 kernel and 2x2 stride
x = MaxPooling2D(pool_size=2, strides=2, padding="same")(x)
```

```
# average pooling with a 2x2 kernel and 2x2 stride
x = AveragePooling2D(pool_size=2, strides=2, padding="same")(x)
```

# Deep learning with Keras

## Réseaux convolutifs > implémentation (i)

Construisons notre propre réseau convolutif:



- Couche 1: 32 filtres 3x3, ReLU, pas de pooling
- Couche 2: 32 filtres 3x3, ReLU, max pooling 2x2 avec stride 2x2
- Couche 3: 16 filtres 3x3, ReLU, max pooling 2x2 avec stride 2x2
- Couche 4: fully connected layer, 32 neurones, ReLU
- Couche 5: fully connected layer, 16 neurones, ReLU
- Couche 6: fully connected layer, 10 neurones, Softmax

# Deep learning with Keras

## Réseaux convolutifs > implémentation (ii)

```
input = Input(shape=[28, 28, 1])

# layer 1
x = Conv2D(32, kernel_size=3, padding="same", activation="relu")(input)

# layer 2
x = Conv2D(32, kernel_size=3, padding="same", activation="relu")(x)
x = MaxPooling2D(pool_size=2, strides=2, padding="same")(x)

# layer 3
x = Conv2D(16, kernel_size=3, padding="same", activation="relu")(x)
x = MaxPooling2D(pool_size=2, strides=2, padding="same")(x)

# fully connected
x = Flatten()(x)
x = Dense(32, activation="relu")(x)
x = Dense(16, activation="relu")(x)
x = Dense(10, activation="softmax")(x)

model = Model(inputs=[input], outputs=[x])
```

# Deep learning with Keras

## Réseaux convolutifs > entraînement et inférence

Les codes d'entraînement et d'inférence sont les mêmes que pour le perceptron multicouche. On doit juste transformer les images en matrice plutôt qu'en vecteur.

On obtient 98,79% d'exactitude !

On pourrait encore raffiner le modèle avec notamment:

- *batch normalization*
- *dropout* dans les couches *fully connected*
- plus de couches
- des connexions résiduelles
- ...

# Transfer learning

Il a été montré qu'on peut entraîner un modèle sur un problème A et l'utiliser sur un problème B.

Par exemple:

- **tâche A:** identifier l'objet principal dans une photo
- **tâche B:** reconnaître des cellules malades ou saines dans des images médicales

Keras permet d'utiliser des modèles profonds pré-entraînés sur la base de données ImageNet. Par exemple, le réseau ResNet 50 couches:

```
from keras.applications import ResNet50
from keras.layers import Dense

resnet = ResNet50((224, 224, 3), weights="imagenet", include_top=False)
out = Dense(n_classes, activation="softmax")(resnet.output)
model = Model(inputs=resnet.input, outputs=out)
```

# Conclusion

Ce qu'on a vu:

- **3 types de modèles**: perceptron binaire, perceptron multicouche et réseaux convolutifs
- **3 fonctions d'activation**: sigmoïde, softmax et ReLU
- **2 fonctions de perte**: entropie croisée binaire et multi-classe
- ...
- et surtout **comment les implémenter** avec TensorFlow/Keras pour traiter un **problème de reconnaissance d'image**

Mais **nous avons juste gratté la surface !**

# Pour aller plus loin !

Améliorer les performances des réseaux:

- régularisation: dropout, weight decay
- conditionnement: batch normalization, deep residual networks
- optimisation: comparaison des méthodes
- transfert: *CS231n: Transfer Learning*

Compréhension:

- fonctions d'activation: *Activation Functions in Neural Networks*
- réseaux convolutifs: *CS231n: Convolutional Neural Networks*
- backpropagation: *What is backpropagation and what is it actually doing?*

Frameworks:

- Quora - How does Caffe 2 compare to TensorFlow?
- Quora - What are the pros and cons of PyTorch vs Keras?

# Merci !

Des questions ?