

Deep Learning par la Pratique

Convolutional Neural Networks et Keras

GIRAUD François-Marie

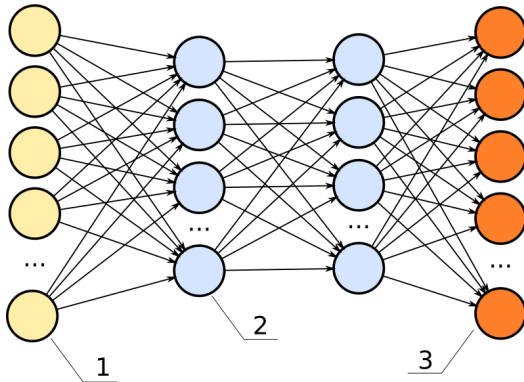


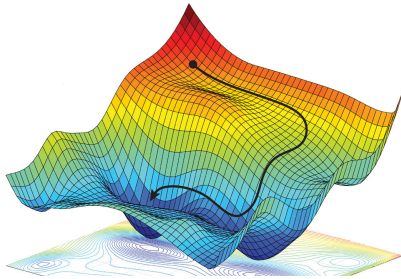
<https://www.orsys.fr/>

Deep Learning par la Pratique

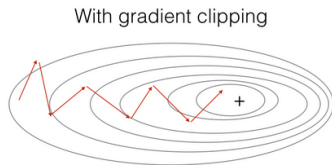
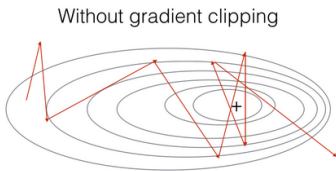
Cours 2 : Keras et Convolutional Neural Networks

Rappels



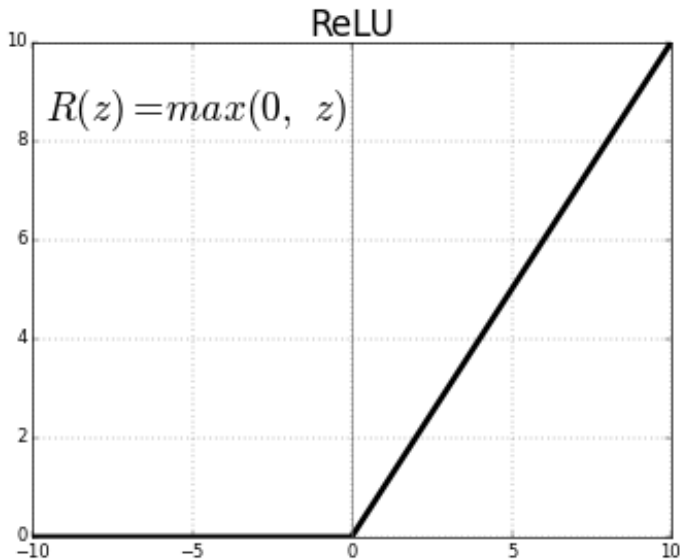


Exploding Gradient \Rightarrow Gradient clipping



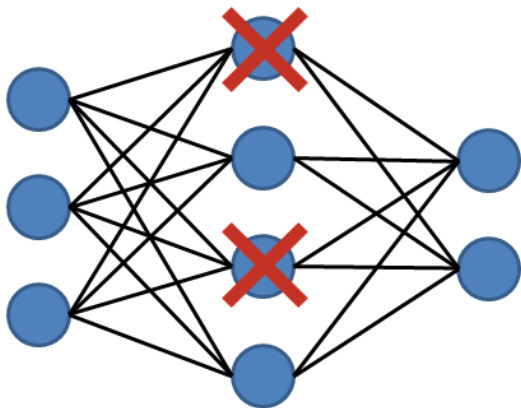
Rappels

Vanishing Gradient \Rightarrow utilisation de ReLu plutôt que les sigmoïdes



Optimisateur plus rapide que SGD

- Root Mean Square Propagation (RMSProp)
- Adaptive Gradient Algorithm (AdaGrad)
- Adaptive Moment Estimation (Adam) \Leftarrow
- ... AdaBound (2019) ?

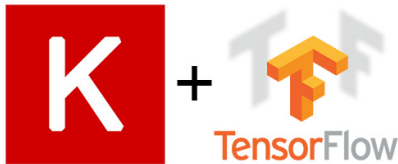


Deep Learning par la Pratique

Introduction à Keras

Introduction à Keras

Keras est une API de haut niveau qui permet de prototyper des réseaux de neurones de toute sorte.



User-friendly :

- Interface simple
- Accès facilité aux métriques d'évaluation

Modulaire :

- Les réseaux se 'branchent' facilement les uns avec les autres
- Tous les réseaux se configurent facilement

Etat de l'art :

- Les modèles et optimiseurs pertinents sont rapidement ajouté à Keras
- Reproduction facile de résultats récents

Facile de développer de nouvelles :

- Couche de réseau (Layers)
- Fonction de perte (Loss)
- Métriques d'évaluation

Réseau de neurones en Keras

```
1 from tensorflow.keras import layers as kl
2
3 model = tf.keras.Sequential()
4 # Il est impératif de spécifier input_shape pour la première couche :
5 model.add(layers.Dense(64, activation='sigmoid', input_shape=(32,)))
6 # Ajouter une autre couche :
7 model.add(layers.Dense(64, activation='relu'))
8 # Une dernière couche de classification avec softmax ;
9 model.add(layers.Dense(10, activation='softmax'))
```

Options de configuration des Layers

```
1  # Utiliser une fonction d'activation :  
2  layers.Dense(64, activation='sigmoid')  
3  # Ou:  
4  layers.Dense(64, activation=tf.keras.activations.sigmoid)  
5  
6  # L1 regularization des poids de la matrice :  
7  layers.Dense(64, kernel_regularizer=tf.keras.regularizers.l1(0.01))  
8  
9  # L2 regularization des biais:  
10 layers.Dense(64, bias_regularizer=tf.keras.regularizers.l2(0.01))  
11  
12 # Initialisation des poids avec une matrice orthogonale :  
13 layers.Dense(64, kernel_initializer='orthogonal')  
14  
15 # Initialisation des biais avec une constante :  
16 layers.Dense(64, bias_initializer=tf.keras.initializers.Constant(2.0))
```


Optimiseur, Loss et Métrique d'évaluation

```
1  #Compilation du modèle
2  model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
3                loss='categorical_crossentropy',
4                metrics=['accuracy'])
5  # Configure a model for mean-squared error regression.
6  model.compile(optimizer=tf.keras.optimizers.Adam(0.01),
7                loss='mse',          # mean squared error
8                metrics=['mae'])    # mean absolute error
9  # Configure a model for categorical classification.
10 model.compile(optimizer=tf.keras.optimizers.RMSprop(0.01),
11               loss=tf.keras.losses.CategoricalCrossentropy(),
12               metrics=[tf.keras.metrics.CategoricalAccuracy()])
```

Affichage du modèle

```
1 model.summary()
```

```
1 Model: "sequential"
```

```
2
```

```
3 Layer (type)
```

```
Output Shape
```

```
Param #
```

```
4
```

```
5 dense (Dense)
```

```
(None, 64)
```

```
2112
```

```
6
```

```
7 dense_1 (Dense)
```

```
(None, 64)
```

```
4160
```

```
8
```

```
9 dense_2 (Dense)
```

```
(None, 10)
```

```
650
```

```
10
```

```
11 Total params: 6,922
```

```
12 Trainable params: 6,922
```

```
13 Non-trainable params: 0
```

```
14
```

```
1 import numpy as np
2
3 data = np.random.random((1000, 32))
4 labels = np.random.random((1000, 10))
5
6 model.fit(data, labels, epochs=10, batch_size=32)
```

```
1  import numpy as np
2
3  data = np.random.random((1000, 32))
4  labels = np.random.random((1000, 10))
5
6  val_data = np.random.random((100, 32))
7  val_labels = np.random.random((100, 10))
8
9  model.fit(data, labels, epochs=10, batch_size=32,
10           validation_data=(val_data, val_labels))
```

```
1 import numpy as np
2
3 data = np.random.random((1000, 32))
4 labels = np.random.random((1000, 10))
5
6 model.fit(data, labels, epochs=10, batch_size=32, validation_split=0.3)
```

```
1  # Instantiates a toy dataset instance:  
2  dataset = tf.data.Dataset.from_tensor_slices((data, labels))  
3  dataset = dataset.batch(32)  
4  
5  model.fit(dataset, epochs=10)  
6  #Or  
7  model.fit(dataset, epochs=10, validation_split=0.3)
```

Evaluation

```
1  # With Numpy arrays
2  data = np.random.random((1000, 32))
3  labels = np.random.random((1000, 10))
4
5  model.evaluate(data, labels, batch_size=32)
6
7  # With a Dataset
8  dataset = tf.data.Dataset.from_tensor_slices((data, labels))
9  dataset = dataset.batch(32)
10
11 model.evaluate(dataset)
```

```
1  1000/1 [=====] - 0s 70us/sample - loss: 199785.4507
2                                - categorical_accuracy: 0.0990
3  32/32 [=====] - 0s 2ms/step - loss: 200361.9849
4                                - categorical_accuracy: 0.0990
```

Evaluation

```
1 result = model.predict(data, batch_size=32)
2 print(result.shape)
```

```
1 (1000, 10)
```


Sauvegarde de la configuration d'un modèle

```
1 import json
2 import pprint
3 # Serialize a model to JSON format
4 json_string = model.to_json()
5 pprint.pprint(json.loads(json_string))
6 # Load a model configuration
7 fresh_model = tf.keras.models.model_from_json(json_string)
```

Sauvegarde des poids d'un modèle

```
1  # Save weights to a HDF5 file  
2  model.save_weights('my_model.h5', save_format='h5')  
3  
4  # Restore the model's state  
5  model.load_weights('my_model.h5')
```

Sauvegarde d'un modèle complet

```
1  # Save entire model to a HDF5 file  
2  model.save('my_model.h5')  
3  
4  # Recreate the exact same model, including weights and optimizer.  
5  model = tf.keras.models.load_model('my_model.h5')
```

Callbacks

```
1  callbacks = [  
2      # Interrompt l'apprentissage si `val_loss`  
3      #ne s'améliore plus depuis 2 epochs  
4      tf.keras.callbacks.EarlyStopping(patience=2, monitor='val_loss'),  
5      # Sauvegarde le meilleur model  
6      tf.keras.callbacks.ModelCheckpoint(filepath='models/bestmodel.hdf5',  
7                                          verbose=1, save_best_only=True)  
8  ]  
9  model.fit(data, labels, batch_size=32, epochs=5,  
10           callbacks=callbacks,  
11           validation_split=0.2)
```

Création d'un Layer

```
1 class MyLayer(layers.Layer):
2     def __init__(self, output_dim, **kwargs):
3         self.output_dim = output_dim
4         super(MyLayer, self).__init__(**kwargs)
5     def build(self, input_shape):
6         # Create a trainable weight variable for this layer.
7         self.kernel=self.add_weight(name='kernel',
8                                     shape=(input_shape[1], self.output_dim),
9                                     initializer='uniform',
10                                    trainable=True)
11    def call(self, inputs):
12        return tf.matmul(inputs, self.kernel)
13    def get_config(self):
14        base_config = super(MyLayer, self).get_config()
15        base_config['output_dim'] = self.output_dim
16        return base_config
17    @classmethod
18    def from_config(cls, config):
19        return cls(**config)
```

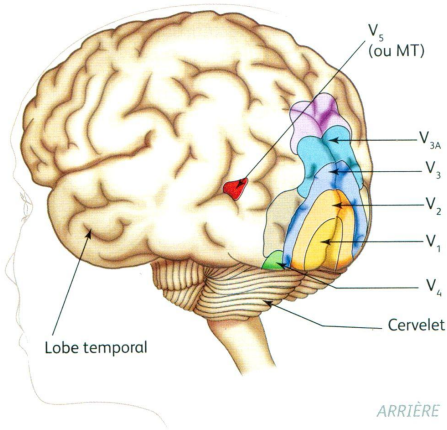
Utilisation de notre nouveau Layer

```
1 model = tf.keras.Sequential()
2 model.add(MyLayer(10))
3 model.add(tf.layers.Activation('softmax'))
4
5 # The compile step specifies the training configuration
6 model.compile(optimizer=tf.keras.optimizers.RMSprop(0.001),
7               loss='categorical_crossentropy',
8               metrics=['accuracy'])
9
10 # Trains for 5 epochs.
11 model.fit(data, labels, batch_size=32, epochs=5)
```

Deep Learning par la Pratique

Convolutional Neural Networks

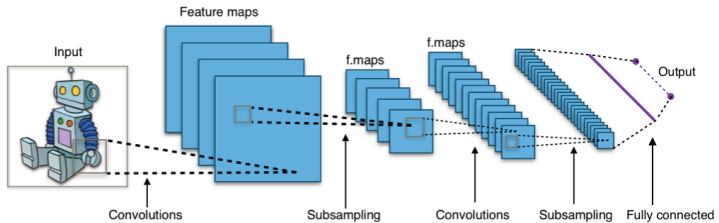
Architecture du cortex visuel



Connexion “en série” de couches

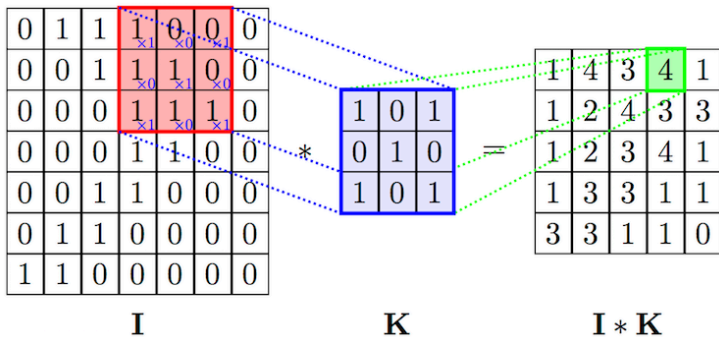
- V1 \approx orientations de lignes
- V2 \approx formes, tailles, couleurs
- V3 \approx aide à la motricité
- V4 \approx reconnaissance d'objets simples
- V5 \approx vitesse des objets
- ...

Convolutional Neural Networks



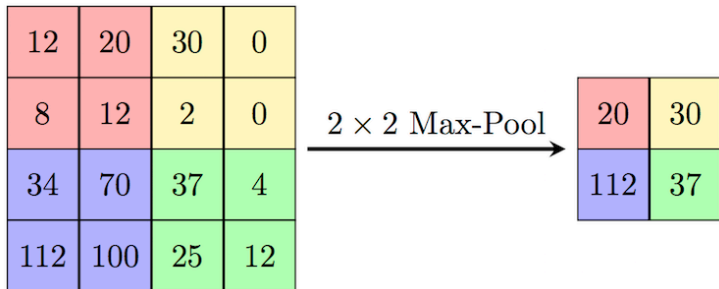
Convolutional Neural Networks

L'opérateur de **convolution**



Convolutional Neural Networks

L'opérateur de **pooling**



Deep Learning par la Pratique

Implémentation d'un CNN avec Keras

Définition d'un modèle CNN

```
1 model = Models.Sequential()
2 #Un premier Layer de 10 convolutions de 3x3 pixels
3 model.add(Layers.Conv2D(10,kernel_size=(3,3),activation='relu',
4                           input_shape=(150,150,3)))
5 #Un Layer "max pooling"
6 model.add(Layers.MaxPool2D(3,3))
7 #Un Layer "Flatten"
8 model.add(Layers.Flatten())
9 #Un Layer "Dense" avec 6 sorties et un softmax
10 model.add(Layers.Dense(6,activation='softmax'))
11 #Compilation du modèle avec la définition de la loss
12 model.compile(optimizer=Optimizer.Adam(lr=0.0001),
13               loss='sparse_categorical_crossentropy',
14               metrics=['accuracy'])
```

Affichage du modèle

```
1 model.summary()
```

```
1 -----
2 Layer (type)                Output Shape                Param #
3 -----
4 conv2d (Conv2D)             (None, 148, 148, 10)       280
5 -----
6 max_pooling2d (MaxPooling2D) (None, 49, 49, 10)         0
7 -----
8 flatten (Flatten)           (None, 24010)              0
9 -----
10 dense (Dense)               (None, 6)                  144066
11 =====
12 Total params: 144,346
13 Trainable params: 144,346
14 Non-trainable params: 0
15 -----
```

Avez-vous des questions ?

Deep Learning par la Pratique

Travaux Pratiques : CNN et Keras

[Keras Tutorial](#)

[Keras-CNN-landscape](#)

[Keras-CNN-emotions](#)