Introduction à l'apprentissage Profond

Said Gounane

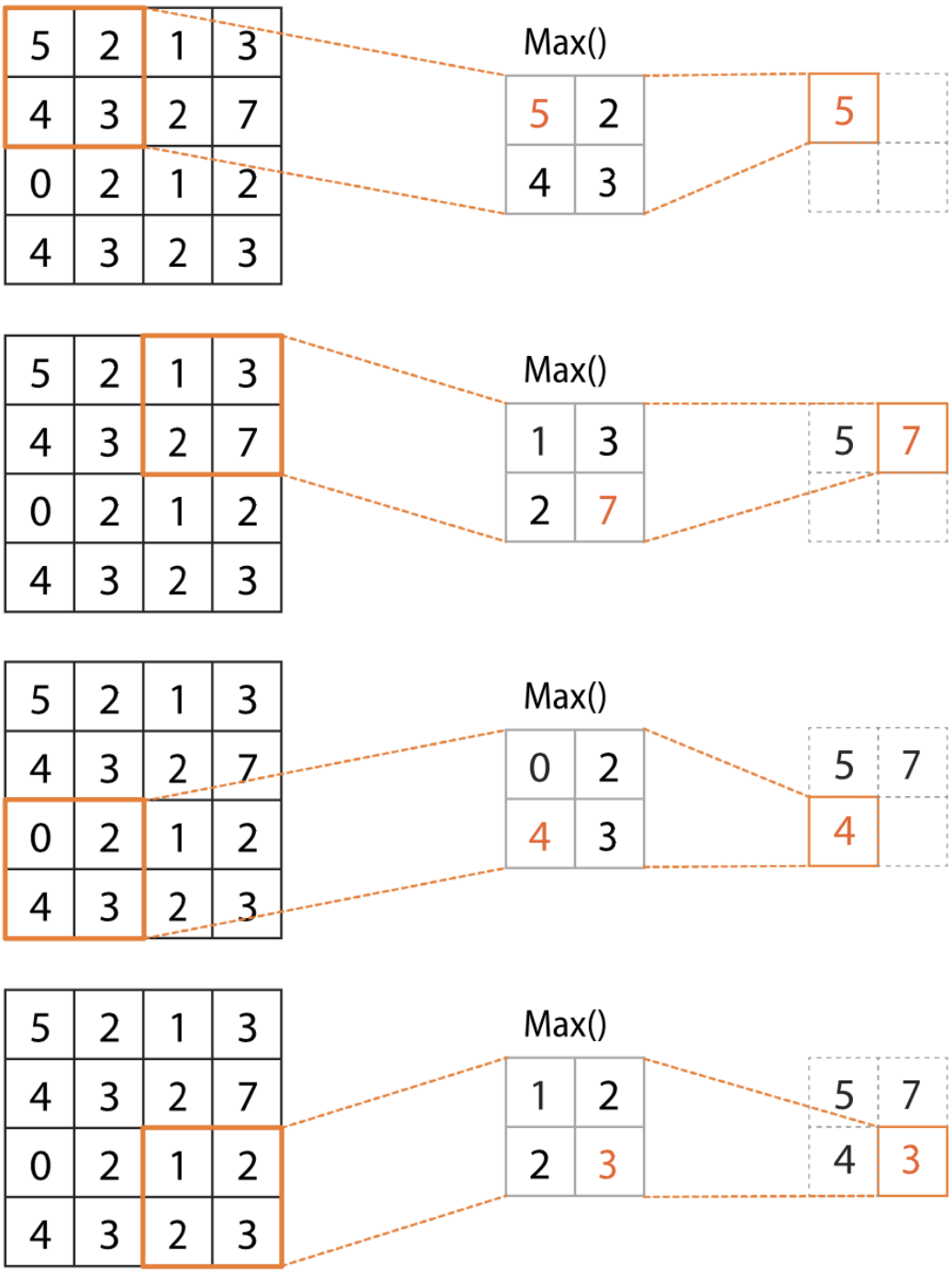
19/02/2022

Contents

1 Reseaux de norones convolutives (CNN)

1.1 1. Introduction

1. Un réseau neuronal convolutif, également connu sous le nom de CNN ou ConvNet, est une classe de réseaux neuronaux spécialisés dans le traitement de données ayant une topologie en forme de grille, telle qu'une image.
2. Une image numérique est une représentation binaire de données visuelles.
3. Il contient une série de pixels disposés en forme de grille qui contient des valeurs de pixel pour indiquer la luminosité et la couleur de chaque pixel.



4. Le cerveau humain traite une énorme quantité d'informations à la seconde où nous voyons une image.

5. Chaque neurone travaille dans son propre champ récepteur et est connecté à d'autres neurones de manière à ce qu'ils couvrent l'ensemble du champ visuel.

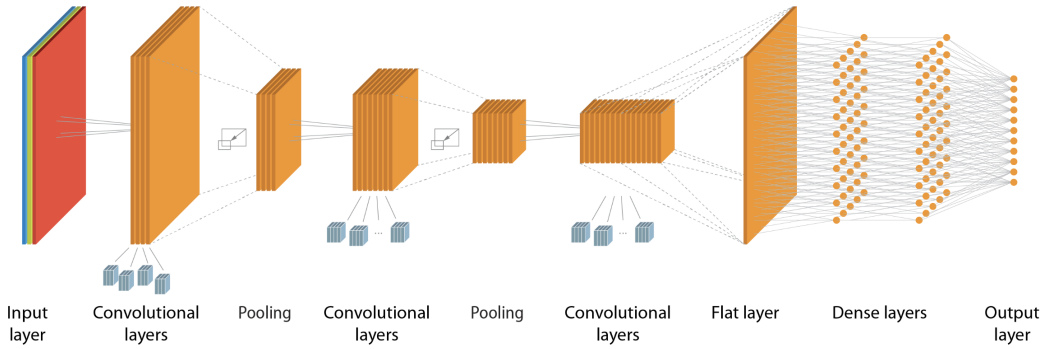
6. Tout comme chaque neurone ne répond aux stimuli que dans la région restreinte du champ visuel appelée champ récepteur dans le système de vision biologique, chaque neurone d'un

CNN traite également les données uniquement dans son champ récepteur.

7. Les couches sont disposées de manière à détecter d'abord les motifs les plus simples (lignes, courbes, etc.) et les motifs les plus complexes (visages, objets, etc.) plus loin. En utilisant un CNN, on peut activer la vue sur les ordinateurs.

1.2 2. Architecture d'un CNN

Un CNN comporte généralement trois couches : 1. une couche convolutive, (convolution) 2. une couche de regroupement (pooling) 3. une couche entièrement connectée.



1.2.1 2.1 Convolution

Cette couche effectue un produit scalaire entre deux matrices: 1. l'ensemble de paramètres apprenables autrement connu sous le nom de noyau 2. la partie restreinte du champ récepteur.

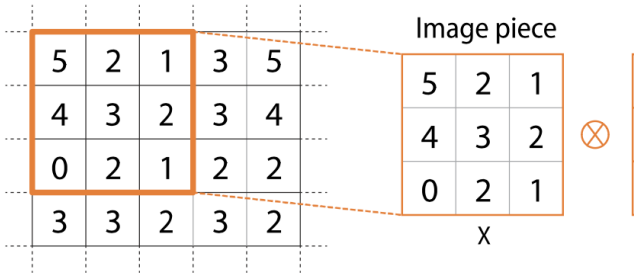


Image piece
5 2 1 3 5
4 3 2 3 4
0 2 1 2 2
3 3 2 3 2
5 2 1
4 3 2
0 2 1
X

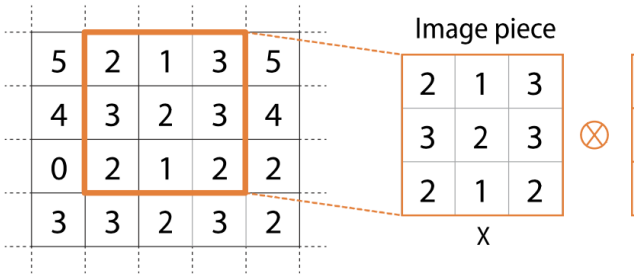


Image piece
5 2 1 3 5
4 3 2 3 4
0 2 1 2 2
3 3 2 3 2
2 1 3
3 2 3
2 1 2
X

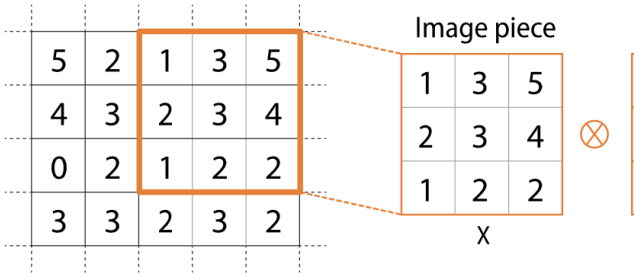
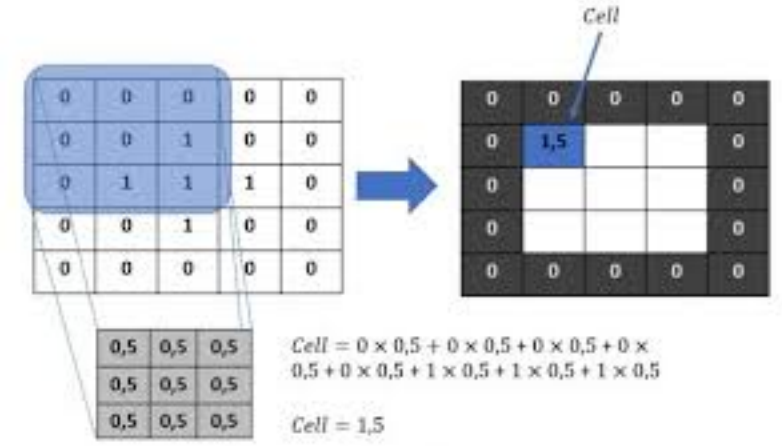
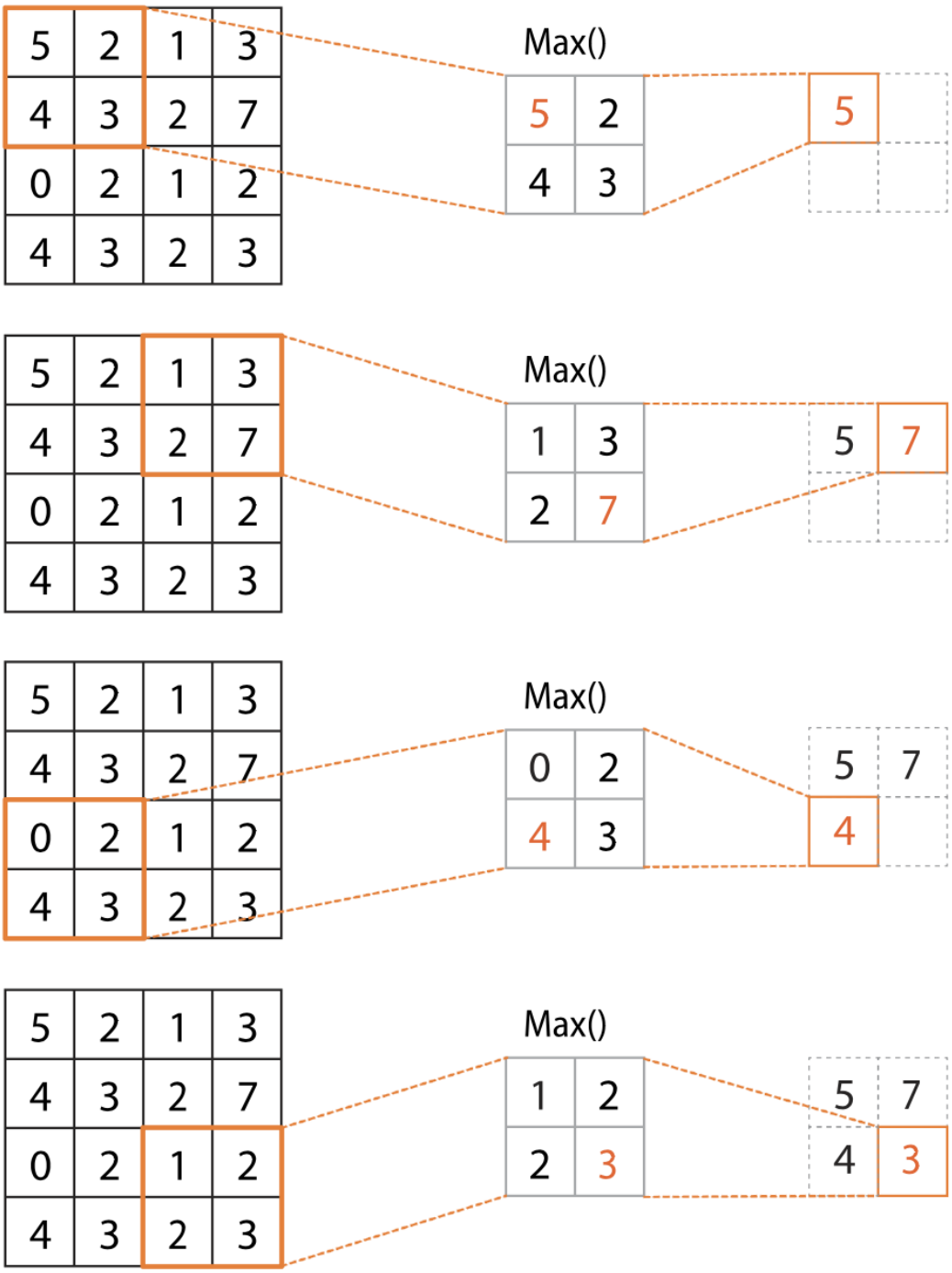


Image piece
5 2 1 3 5
4 3 2 3 4
0 2 1 2 2
3 3 2 3 2
1 3 5
2 3 4
1 2 2
X

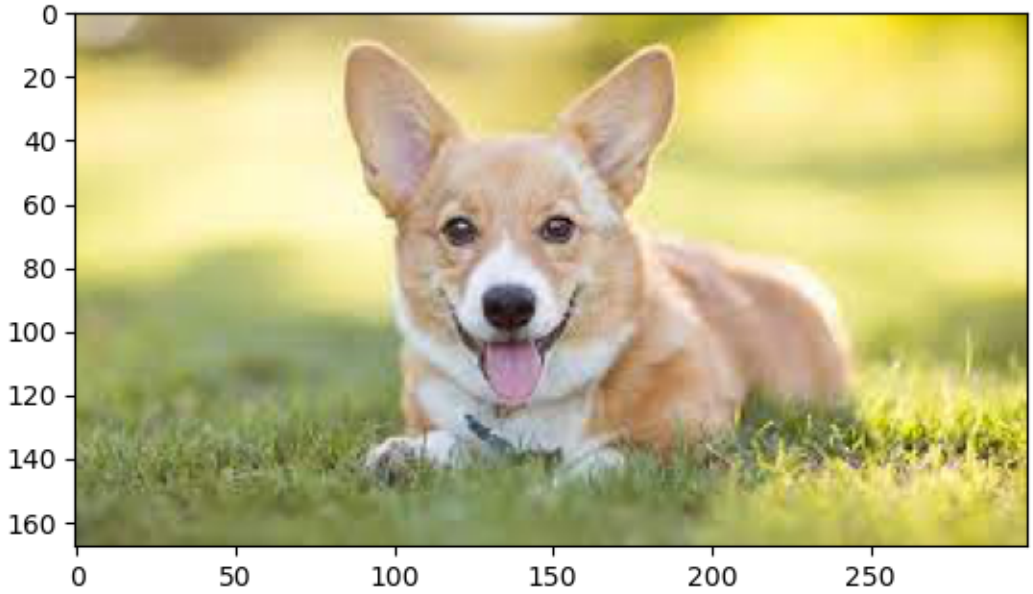

Cell
0 0 0 0 0
0 0 1 0 0
0 1 1 1 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
0 1,5 0 0 0
0 0 0 0 0
0 0 0 0 0
0,5 0,5 0,5
0,5 0,5 0,5
0,5 0,5 0,5
Cell = 0 × 0,5 + 0 × 0,5 + 0 × 0,5 + 0 × 0,5 + 0 × 0,5 + 1 × 0,5 + 1 × 0,5 + 1 × 0,5
Cell = 1,5



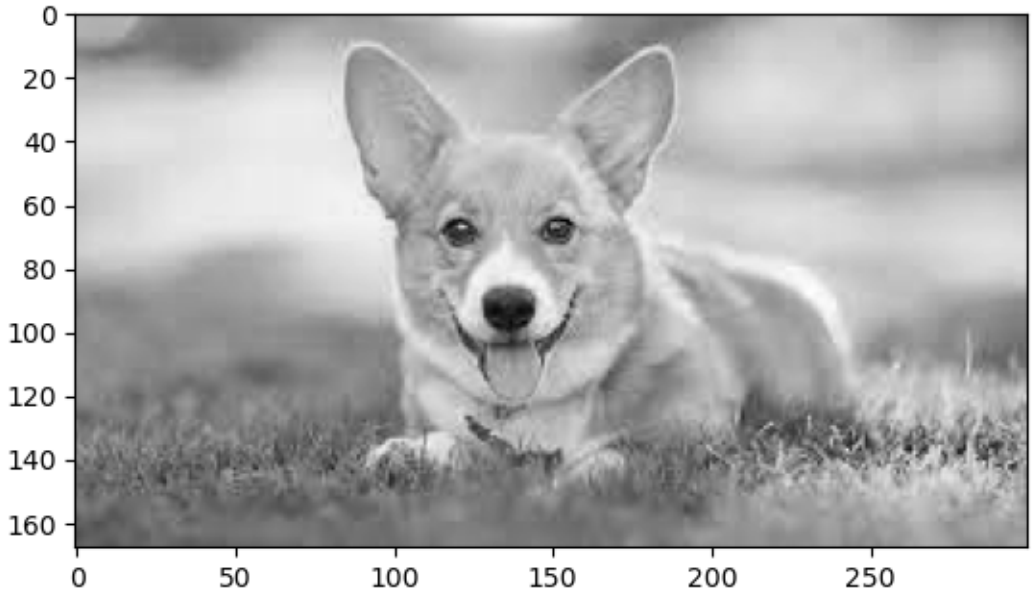
[2]:
```
import numpy as np
from matplotlib import image
from matplotlib import pyplot
img = image.imread('koala.jpg')
print(img.dtype)
print(img.shape)
```

```
pyplot.imshow(img)
pyplot.show()
```

```
uint8
(168, 300, 3)
```



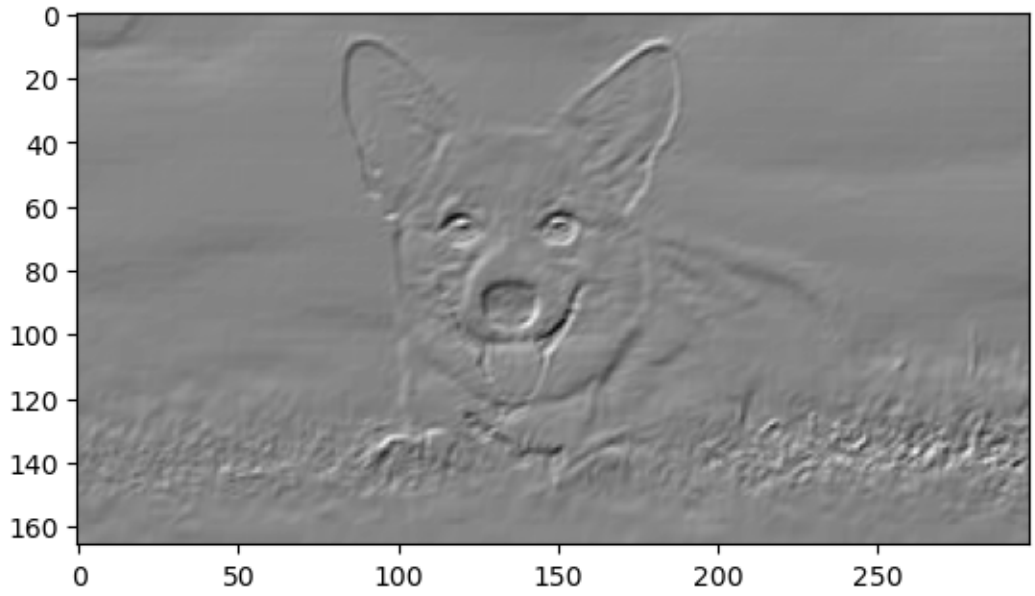
[5]:
```
gimg=np.average(img, axis=2).astype(int)
pyplot.imshow(gimg,cmap='gray')
pyplot.show()
```



[6]:
```
import matplotlib.pyplot as plt
kernel=np.array([[-1, -2, -1],[-2,0,2],[1,2,1]])
def convolution(img,kernel):
    n,m=img.shape
    r=np.zeros((n-2,m-2))
    for i in range(1,n-1):
        for j in range(1,m-1):
            r[i-1,j-1]=np.sum(gimg[i-1:i+2,j-1:j+2]*kernel)
    return r

r=convolution(gimg,kernel)
plt.imshow(r,cmap="gray")
```

[6]:
```
<matplotlib.image.AxesImage at 0x126a4d550>
```

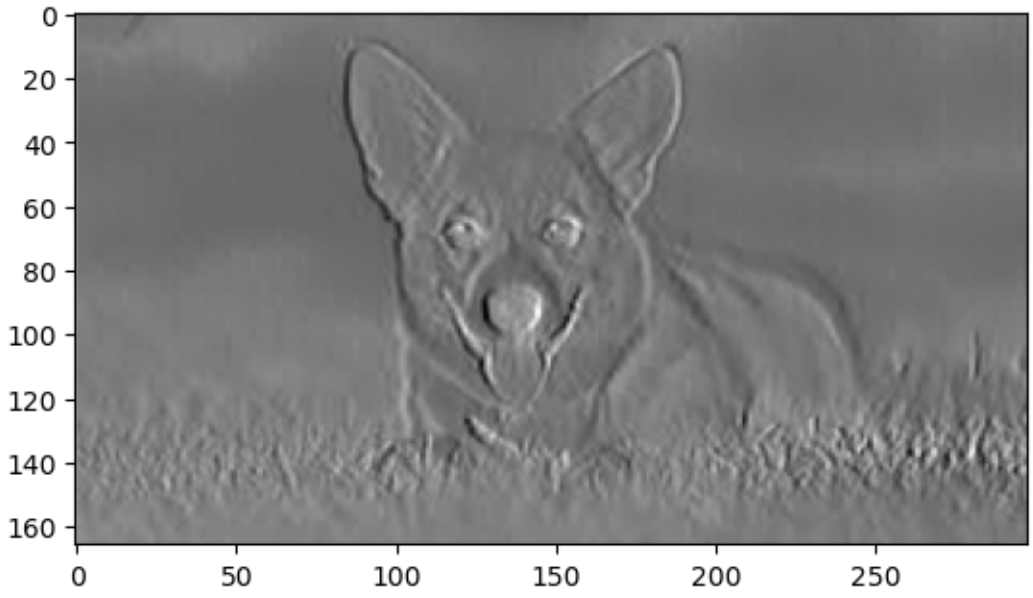


[8]:
```
kernel=np.array([[-1,0,1],[-2,0,2],[-1,0,0]])
def convol(img,kernel):
    k=len(kernel)//2
    print(k)
    r=np.zeros((img.shape[0]-k-1,img.shape[1]-k-1))
    for i in range(k,img.shape[0]-k-1):
        for j in range(k,img.shape[1]-k-1):
            f=img[i-k:i+k+1,j-k:j+k+1]
            r[i,j]=np.sum(f*kernel)
```

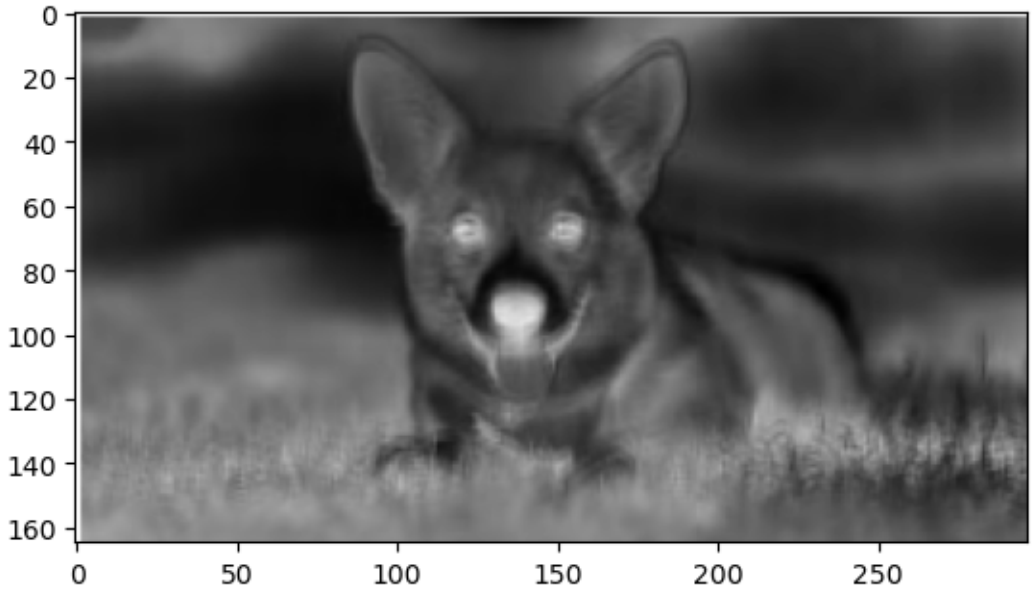
```
    return r
r=convol(gimg,kernel)
pyplot.imshow(r,cmap=pyplot.get_cmap('gray'))
pyplot.show()

kernel=np.
 ↪array([[-1,-1,-1,0,0],[-1,0,0,0,0],[-1,0,0,0,0],[-1,0,0,0,0],[-1,-1,-1,0,0]])
r=convol(gimg,kernel)
pyplot.imshow(r,cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

1



2



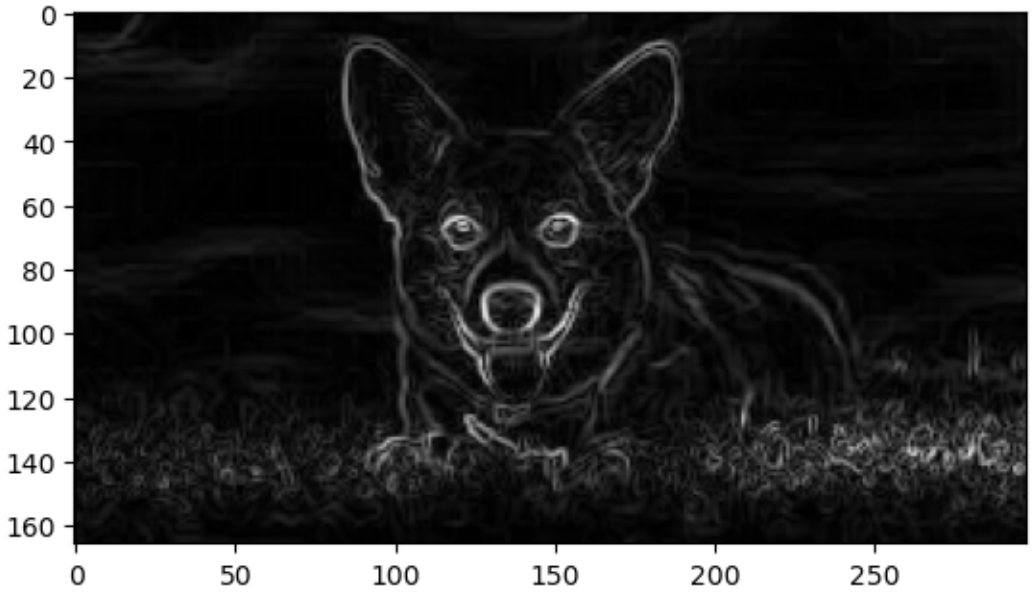
1.2.2 Sobel Edge detection

[10]:
```
kernel=np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
rx=convol(gimg,kernel)
pyplot.imshow(rx/np.max(rx),cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

1

[11]:
```
ry=convol(gimg,kernel.T)
r=np.sqrt(rx*rx+ry*ry)
r=r/np.max(r)
pyplot.imshow(r,cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

1



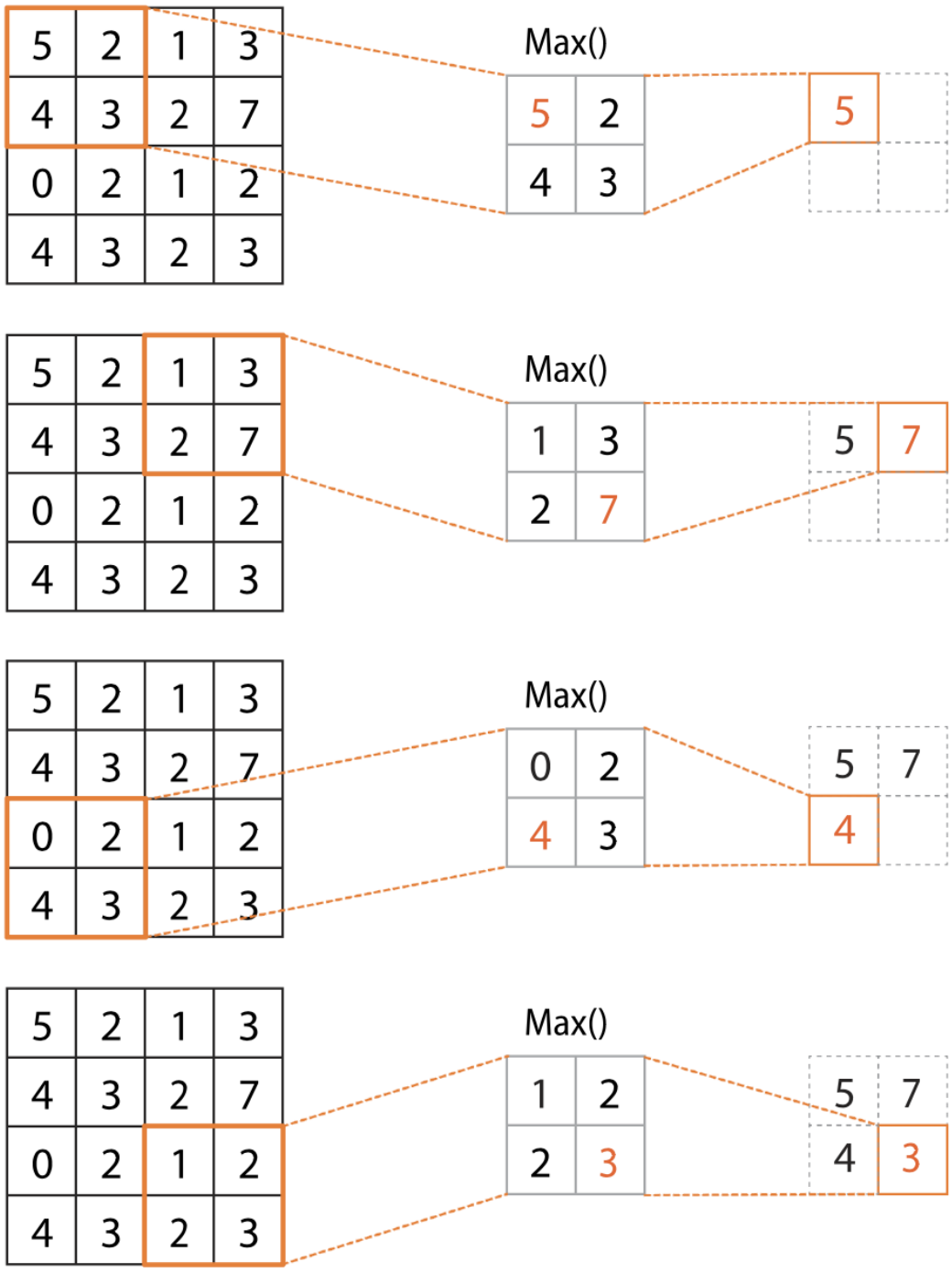
1.3 2.2 Pooling (regroupemen)

La couche de regroupement remplace la sortie du réseau à certains emplacements en dérivant une statistique récapitulative des sorties à proximité.

Cela aide à réduire la taille spatiale de la représentation, ce qui diminue la quantité requise de calculs et de pondérations.

L'opération de regroupement est traitée sur chaque tranche de la représentation individuellement.

Il existe plusieurs fonctions de regroupement telles que: 1. la moyenne du voisinage rectangulaire, 2. la norme L2 du voisinage rectangulaire 3. la moyenne pondérée basée sur la distance par rapport au pixel central. 4. Max pooling



Cependant, le processus le plus populaire est la mise en commun maximale, qui signale la sortie maximale du voisinage.

[62]:
```
## Fonction pour calculer le pooling  de deux matrices M et K
def maxpooling(img,n):
```

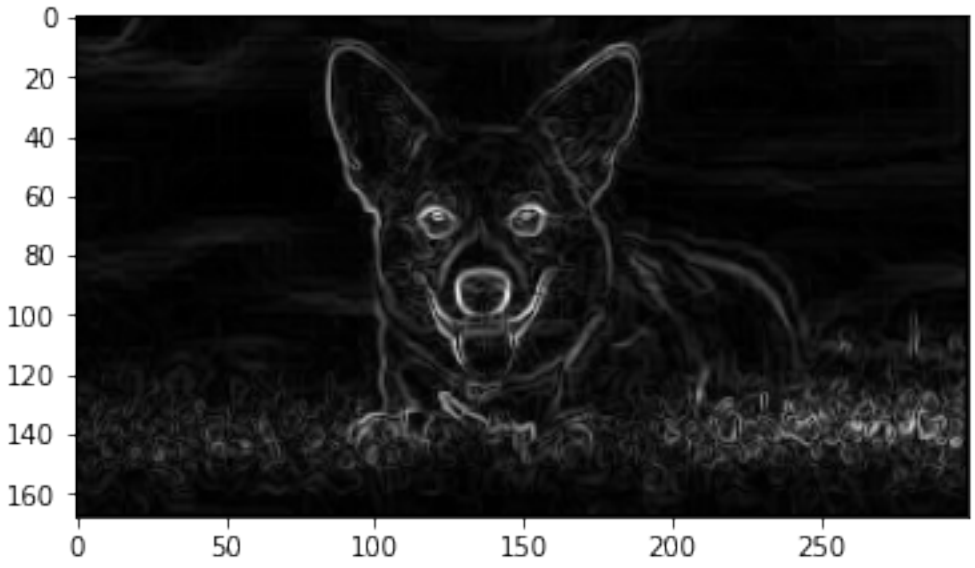
```
    return r
```

[63]:
```
m=maxpooling(r,4)
print(m.shape,gimg.shape)
```

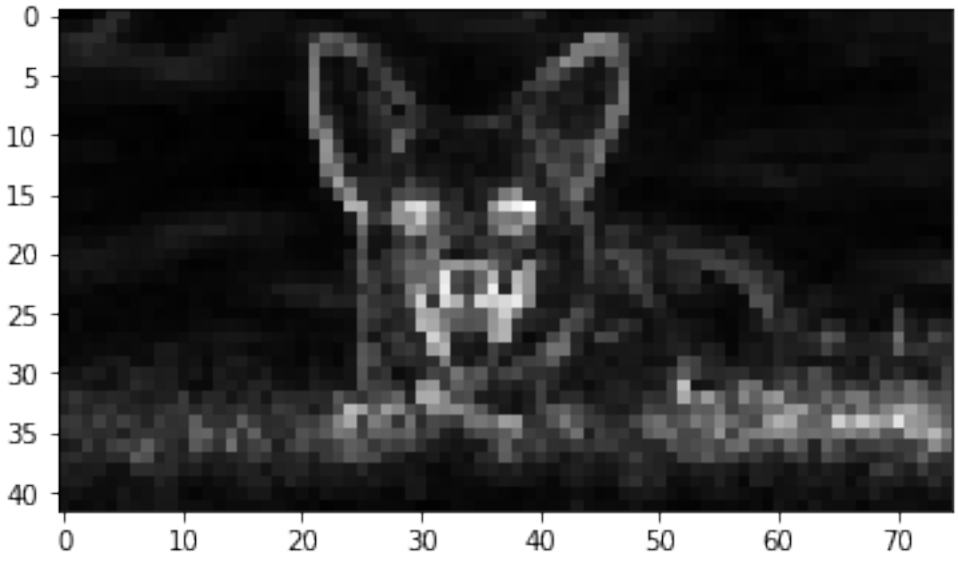
```
168
(42, 75) (168, 300)
```

[64]:
```
print(r.size)
pyplot.imshow(r,cmap=pyplot.get_cmap('gray'))
pyplot.show()
print(m.size)
pyplot.imshow(m,cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

```
50400
```




```
3150
```



1.3.1 3. Architecture d'un réseau de neurones convolutif

- Un CNN est simplement un empilement de plusieurs couches de convolution, pooling, correction ReLU et fully-connected.
- Chaque image reçue en entrée va donc être filtrée, réduite et corrigée plusieurs fois, pour finalement former un vecteur. Dans le problème de classification, ce vecteur contient les probabilités d'appartenance aux classes
- Tous les réseaux de neurones convolutifs doivent commencer par une couche de convolution et finir par une couche fully-connected.
- Plus il y a de couches, plus le réseau de neurones est "profond" (Deep Learning)

1.3.2 3.1 Paramétrage des couches

- Les couches de convolution et de pooling possèdent des hyperparamètres. La taille des feature maps en sortie des couches de convolution et de pooling dépend de ces hyperparamètres.

Paramétres de Couche d'entré Chaque image (ou feature map) est de dimensions $ W×H×D $, où - **W** est sa largeur en pixels, - **H** sa hauteur en pixels et - **D** le nombre de canaux (1 pour une image en noir et blanc, 3 pour une image en couleurs).

Paramétres de Couche de convolution

- **K** Nombre de filtres
- **F** Taille du filtre , chaque filtre est de dimensions FFD pixels.
- **S** Le pas du mouvement du filtre (strides)
- **P** Le zero-padding, on ajoute à l'image en entrée de la couche un contour noir d'épaisseur P pixels. Sans ce contour, les dimensions en sortie sont plus petites

Ainsi, la matrice en sortie de la couche de convolution est de dimension $W_c \times H_c \times D_c$ avec:

$$W_c = \frac{W - F + 2P}{S} + 1$$

$$H_c = \frac{H - F + 2P}{S} + 1$$

$$D_c = K$$

En general en prend F=3,P=1,S=1 ou F=5,P=2,S=1 #### Paramétres de Couche de Pooling - **F** l'image est découpée en cellules carrées de taille **F×F** pixels - **S** les cellules sont séparées les unes des autres de S pixels

Ainsi, la matrice en sortie de la couche de pooling est de dimension $W_p \times H_p \times D_p$ avec:

$$W_p = \frac{W - F}{S} + 1$$

$$H_p = \frac{H - F}{S} + 1$$

$$D_p = D$$

En general en prend F=2 et S=2

1.4 4. Application: keras

1.4.1 4.1 Réseau de neurone simple

Création d'un réseau de neurones vide mdl = Sequential()
Ajout de une couche fully-connected, avce une fonction d'activation ReLU mdl.add(Dense(20, activation='relu'))

Ajout de la dernière couche fully-connected mdl.add(Dense(10, activation='softmax'))

Compilation du model mdl.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.legacy.Adam(),metrics=['accuracy'])

mdl.fit(X_train,Y_train_one_hot, batch_size=10, epochs=4,validation_split = 0.2, verbose=1)

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
from tensorflow.keras.utils import to_categorical
```

[11]:
```
import pandas as pd
(X_train,y_train), (X_test,y_test) = mnist.load_data()
```

```
Y_train_one_hot = to_categorical(y_train)
Y_test_one_hot = to_categorical(y_test)
```

[9]:
```
## Création d'un reseau de neurone dense avec deux couches cachée de 200 et 100␣
↪neurones
## et une couche de sortie 10 neurones
#Model: "sequential_8"
#_________________________________________________________________
# Layer     (type)            Output Shape              Param #
#=================================================================
# dense_15 (Dense)            (10, 100)                 78500
#
# dense_16 (Dense)            (10, 10)                  1010
#
#=================================================================
```

[12]:
```
## Performances du modèle
```

1.4.2 Fashion MNIST

Création d'un réseau de neurones vide mdl = Sequential()

Ajout de la première couche de convolution, suivie d'une couche ReLU mdl.add(Conv2D(20, (3, 3), input_shape=(28, 28, 1), padding='same', activation='relu'))

Ajout de la deuxième couche de convolution, suivie d'une couche ReLU mdl.add(Conv2D(20, (3, 3), padding='same', activation='relu'))

Ajout de la première couche de pooling mdl.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))

Conversion des matrices 3D en vecteur 1D mdl.add(Flatten())

Ajout de la première couche fully-connected, suivie d'une couche ReLU mdl.add(Dense(20, activation='relu'))

Ajout de la deuxième couche fully-connected, suivie d'une couche ReLU mdl.add(Dense(20, activation='relu'))

Ajout de la dernière couche fully-connected qui permet de classifier mdl.add(Dense(10, activation='softmax'))

[14]:
```
from tensorflow import keras
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D,␣
↪MaxPooling2D
from tensorflow.keras.models import Sequential
```

```
from tensorflow.keras.utils import to_categorical
```

[15]:
```
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
```

[16]:
```
#Normalisation des données
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
```

[17]:
```
#Adaptation des données au modèle

train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
```

[23]:
```
#Créer un CNN avec
#_________________________________________________________________
#Layer type            Output Shape              Param
#=================================================================
# conv2d              (None, 26, 26, 64)            640
#
# activation          (None, 26, 26, 64)              0
#
# max_pooling2d       (None, 13, 13, 64)              0

# flatten             (None, 10816)                   0

# dense               (None, 10)                  108170

# activation          (None, 10)                    0

#=================================================================
#Total params: 108,810
#Trainable params: 108,810
#Non-trainable params: 0
#_________________________________________________________________
```

[ ]:
```
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=10)
```

[33]:
```
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.2808 -
accuracy: 0.8995
Test loss 0.280762255191803
```

```
Test accuracy 0.8995000720024109
```

[62]:
```
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
```

```
313/313 [==============================] - 1s 3ms/step
9
```

[ ]:

2 Les réseaux de neurones récurrents

2.1 1. Introduction

Dans la première partie de ce cours, nous avons introduit les réseaux de neurones feedforward, dans lesquels l'information transite des entrées vers les sorties. - Les perceptrons multicouches (PMC, ou MLP pour MultiLayer Perceptrons, en anglais) - Les réseaux convolutionnels (CNN) rentrent dans cette catégorie.

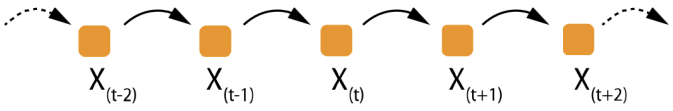
Ces modèles sont la référence pour l'apprentissage par machine, car ils ont de nombreux avantages : - performances; - rapides en décision (mais longs en apprentissage) ; - supportent très bien des hautes dimensions en entrée et/ou en sortie ; - capables d'extraire des caractéristiques automatiquement, notamment sur les images avec les CNN ; - existence d'un algorithme efficace d'apprentissage : la rétropropagation du gradient.

Limites liées à la taille des données - la taille de l'entrée et la taille de sortie doivent être identiques pour tous les exemples à traiter. - Dans la pratique, dans la plupart des domaines où l'on est susceptible d'utiliser des réseaux de neurones, cette contrainte n'est pas vérifiée

On a besoin d'un modèle capable de prendre en compte des données de taille variable, aussi bien en entrée qu'en sortie.

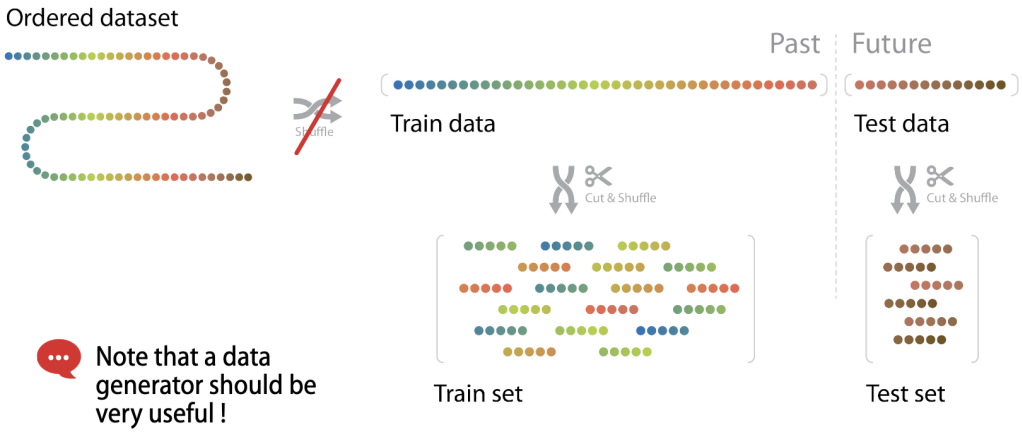
2.2 2. Les réseaux de neurones récurrents

Les réseaux de neurones récurrents (ou RNN pour Recurrent Neural Networks) sont une catégorie de réseaux de neurones dédiée au traitement de séquences, c'est-à-dire aux signaux de taille variable.



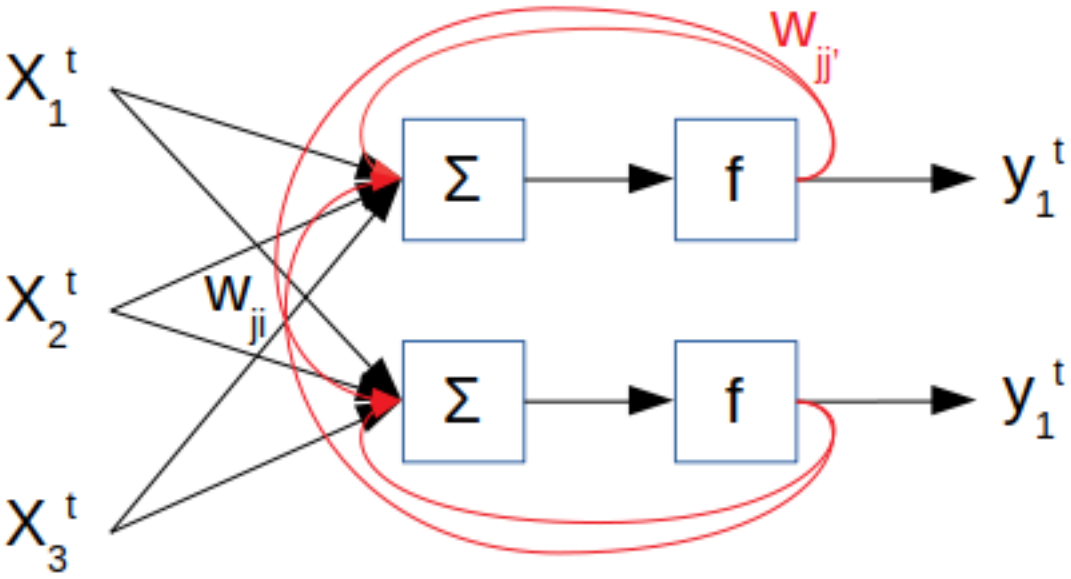
Les signaux d'entrée x qui varient au cours du temps, seront donc notées x^t , t étant la position de la fenêtre glissante dans la séquence d'entrée (voir figure).

Preparation of sequence data



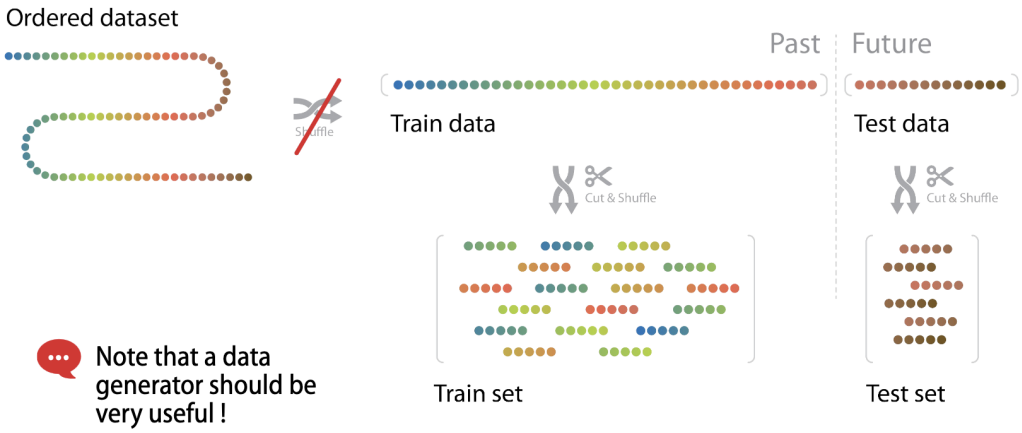
$$y_i^t = f(\sum_j W_{ji} x_i^t)$$

Les réseaux de neurones récurrents reposent sur deux principes : - le premier principe est l'astuce de la fenêtre glissante, qui permet de traiter des signaux de taille variable ; - le second principe est l'utilisation de connexions récurrentes qui permettent d'analyser la partie passée du signal.

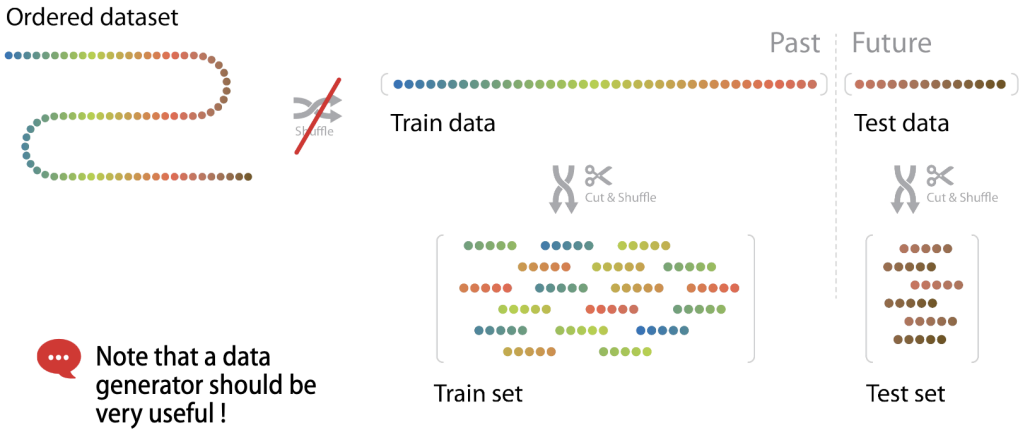


$$y_i^t = f(\sum_j W_{ji} x_i^t + \sum_{j'} R_{j'i} y_i^{t-1})$$

Preparation of sequence data

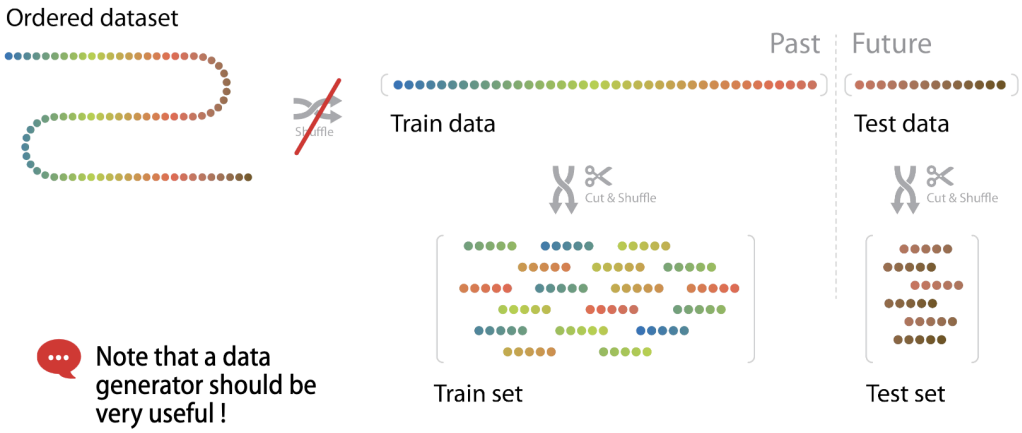


Preparation of sequence data

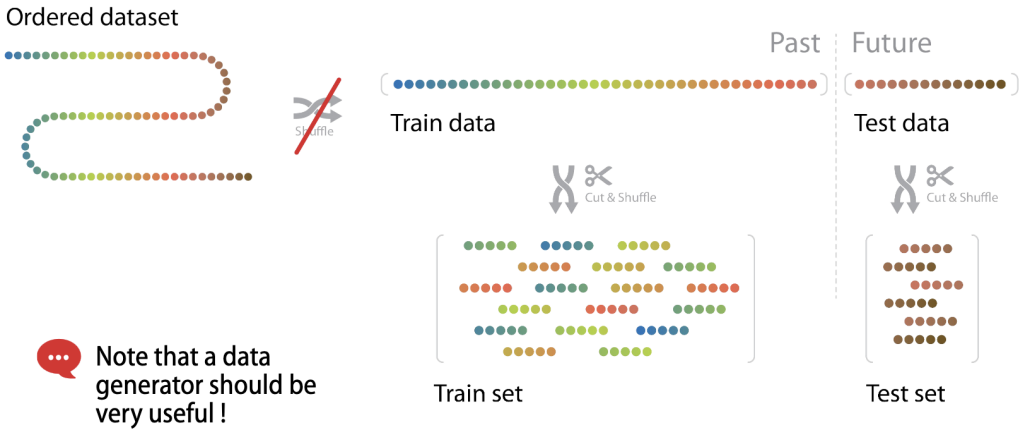


source: https://cloud.univ-grenoble-alpes.fr/index.php/s/wxCztjYBbQ6zwd6?dir=undefined&openfile=565851274

Preparation of sequence data



Preparation of sequence data



[2]:
```
import numpy as np
def sigmoid(x):
    return 1/(1+np.exp(-x))
class RnnLayer:
    def __init__(self, n_units, input_size, activation):
        self.input_size=input_size
        self.activation=activation
        self.Wx=np.random.randn(input_size,n_units)
        self.Wy=np.random.randn(n_units,n_units)
        self.h=np.zeros((n_units,1))
        self.b=np.random.randn(n_units,1)
    def forword(self,X):
        for x in X:
```

```
            A=self.Wx.T.dot(x.reshape((self.input_size,1)))
            B=self.Wy.T.dot(self.h)
            self.h=self.activation(A+ B+self.b)
        return self.h

r=RnnLayer(3,2,sigmoid)
X=np.array([[1,1],[1,2],[1,3],[1,2],[1,3]])
print(X.shape)
y1=r.forword(X)
print(y1)
```

```
(5, 2)
[[0.99149139]
 [0.03531413]
 [0.05861744]]
```

[3]:
```
# SimpleRNN model
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN

#32 serie de 10 exemples de (1,8)
inputs = np.random.random([32, 5, 2]).astype(np.float32)

#Créer une couche de 4 neurones recurent
simple_rnn = SimpleRNN(3)

output = simple_rnn(inputs[0].reshape((1,5,2)))  # The output has shape `[32,␣
 ↪4]`.
print(output.shape)
```

```
(1, 3)
```

[4]:
```
simple_rnn = SimpleRNN(4,return_sequences=True, return_state=True)

# whole_sequence_output has shape `[32, 10, 4]`.
# final_state has shape `[32, 4]`.
whole_sequence_output, final_state = simple_rnn(inputs)
print(whole_sequence_output.shape)
```

```
(32, 5, 4)
```

[6]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Créer une serie de nombre x de N éléments [x1, x2, ...., xN]
t=np.arange(0,10,0.1)
```

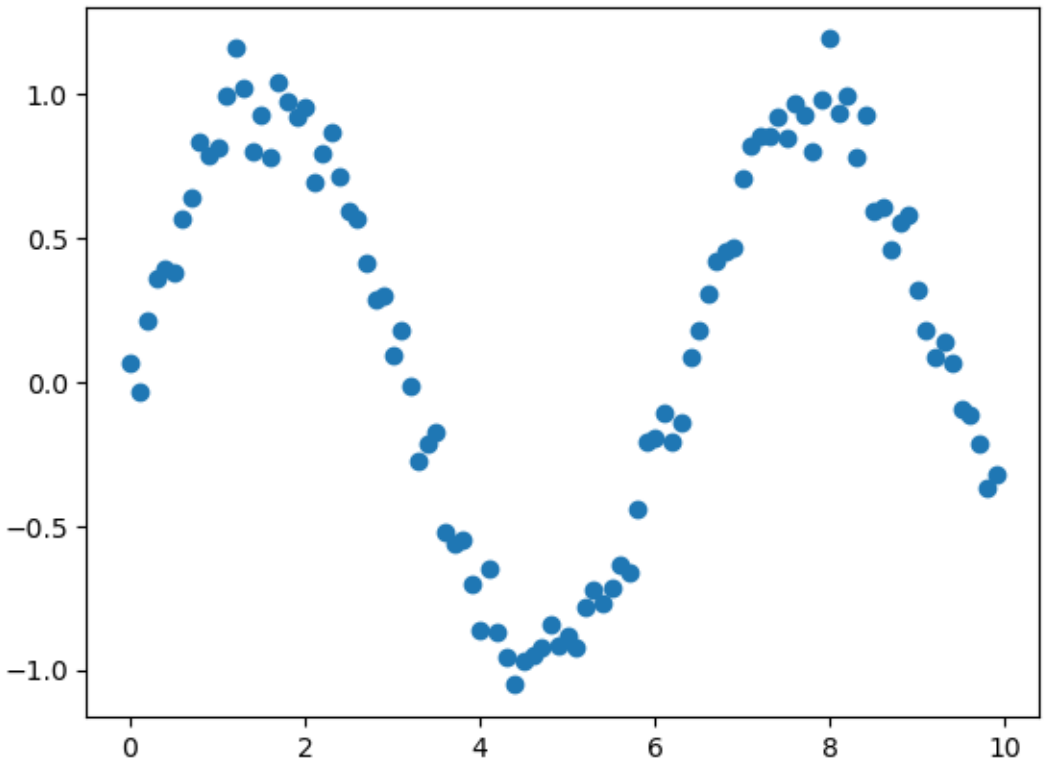
```
print(t.shape)
x=np.sin(t)+np.random.randn(len(t))*0.1
n=len(x)
plt.scatter(t,x)

# Transformer cette serie sous forme de plusieurs sequeces de s éléments
# [x1, x2, ...., xs] y1=X(s+1), [x2, x3, ...., x(s+1)] y2=x(s+2), ....

s=6
X=[]
Y=[]
for i in range(n-s):
    X.append(x[i:i+s])
    Y.append([x[i+s]])
X=np.array(X)
Y=np.array(Y)

# créer x_train , y_train et x_test et y_test
x_train,x_test=X[:80,:],X[80:100,:]
y_train,y_test=Y[:80,:],Y[80:100,:]
print(X.shape)
```

```
(100,)
(94, 6)
```



[13]:
```
# SimpleRNN model
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN

model = Sequential()
model.add(SimpleRNN(units=20, input_shape=(s,1), activation="relu"))
model.add(Dense(5, activation="relu"))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='rmsprop')
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_3 (SimpleRNN)    (None, 20)                440

 dense_2 (Dense)             (None, 5)                 105

 dense_3 (Dense)             (None, 1)                 6

=================================================================
Total params: 551
Trainable params: 551
Non-trainable params: 0
_________________________________________________________________
```

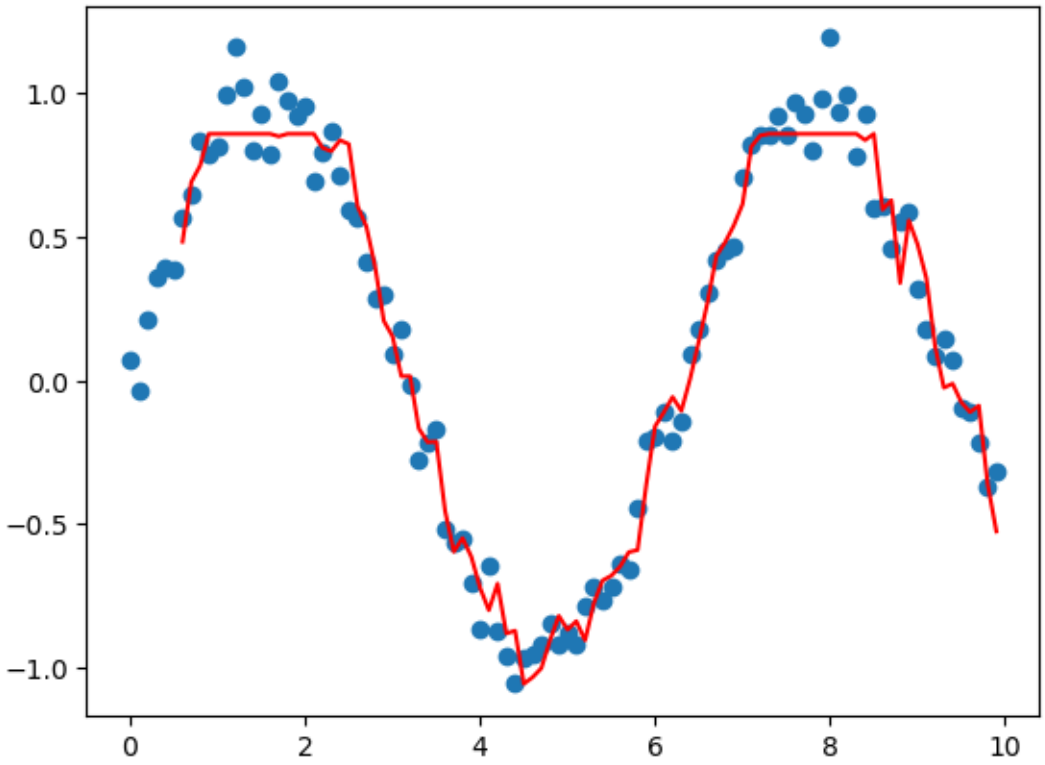
[14]:
```
model.fit(x_train,y_train, epochs=200, batch_size=16, verbose=0)
trainPredict = model.predict(x_train)
testPredict= model.predict(x_test)
predicted=np.concatenate((trainPredict,testPredict),axis=0)
```

```
3/3 [==============================] - 0s 1ms/step
1/1 [==============================] - 0s 12ms/step
```

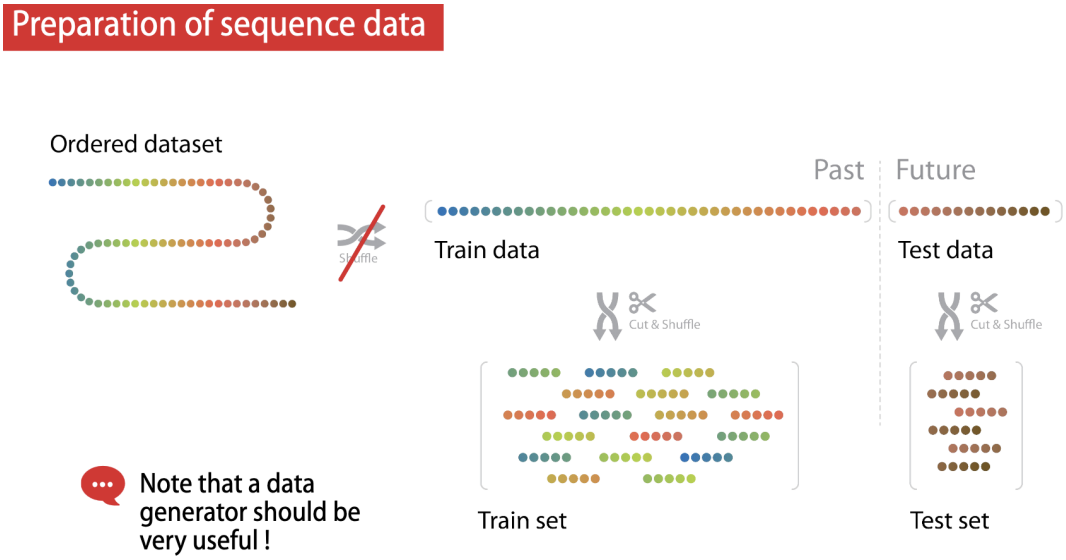
[15]:
```
trainScore = model.evaluate(x_train, y_train, verbose=0)
print(trainScore)
```

```
0.012087366543710232
```

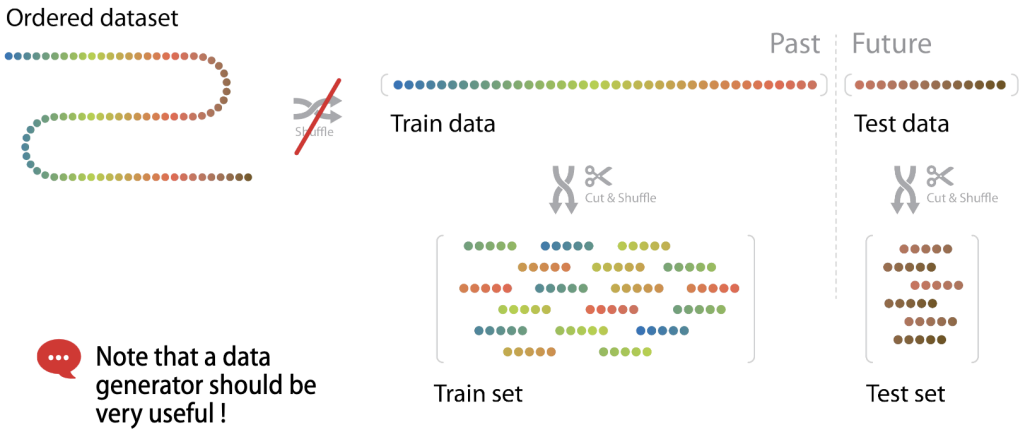
[16]:
```
plt.scatter(t,x)
plt.plot(t[:94]+0.1*s,predicted, color="red")
#plt.axvline(df.index[Tp], c="r")
plt.show()
```



Recall backpropagation Backpropagation through time - exploding gradient -> gradient cliping - vannashing gradient -> activation func, weight init, net architecture



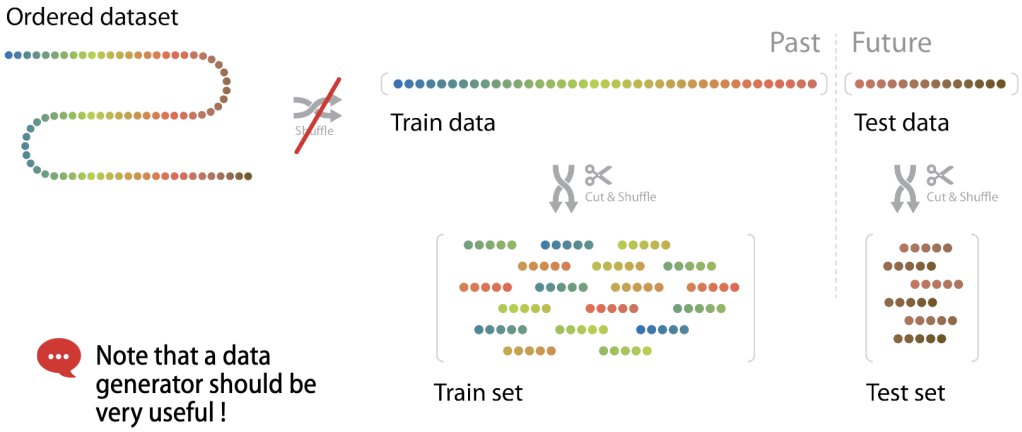
Preparation of sequence data



[78]:
```
from tensorflow.keras.layers import LSTM
inputs=tf.random.normal([32,20,8])
print(x.shape)
lstm=tf.keras.layers.LSTM(16)
output=lstm(inputs)
print(output.shape)
```

```
(32, 20, 8)
(32, 16)
```

Preparation of sequence data



[79]:
```
lstm=tf.keras.layers.LSTM(16,return_sequences=True,return_state=True)
output, memoty_state, carry_state=lstm(inputs)
print(output.shape)
```

```
(32, 20, 16)
```

Preparation of sequence data