Examen 2: Algorithmes d'apprentissage profond

- 420-A60-BB Neila Mezghani
- Author: Ricardo Vallejo

1. Téléchargez le contenu de la base de données (utilisez tf.keras.datasets.cifar10).

In [152…

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Load the data
cifar10 = tf.keras.datasets.cifar10
```

2. La base de données est répartis en des données d’entrainement et des données de test. Formez les deux sous-ensembles de données x_trainet x_test correspondant respectivement aux données d’entrainementet de test

In [153…

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

3. Vérifiez la dimension des données d’entrainement et de test, le nombre de classes et le nombre d’échantillons par classe.

In [154…

```
# Shape of données d’entrainement
input_train_shape = x_train.shape
input_train_shape
```

Out[154…

```
(50000, 32, 32, 3)
```

In [155…

```
# La dimension de donnes de entrainement est 50000 images, de 32x32 pixels et 3 channel
```

In [156…

```
# Shape of données des tests
input_test_shape = x_test.shape
input_test_shape
```

Out[156…

```
(10000, 32, 32, 3)
```

In [157…

```
# La dimension de donnes de entrainement est 10000 images, de 32x32 pixels et 3 channel
```

In [158…

```
import pandas as pd
```

```
Y_traindf = pd.DataFrame(y_train, columns = ['class'])
Y_traindf.groupby('class').size()
```

Out[158…

```
class
0    5000
1    5000
2    5000
3    5000
4    5000
5    5000
6    5000
7    5000
8    5000
9    5000
dtype: int64
```

In [159…

```
# Le numero de classes est 10, et ilya 5000 echantillons par class dans le jeux de donn
```

In [160…

```
Y_testdf = pd.DataFrame(y_test, columns = ['class'])
Y_testdf.groupby('class').size()
```

Out[160…

```
class
0    1000
1    1000
2    1000
3    1000
4    1000
5    1000
6    1000
7    1000
8    1000
9    1000
dtype: int64
```

In [161…

```
# Le numero de classes est 10, et ilya 1000 echantillons par class dans le jeux de donn
```

4. Affichez une dizaine d’échantillons de la base de données

In [162…

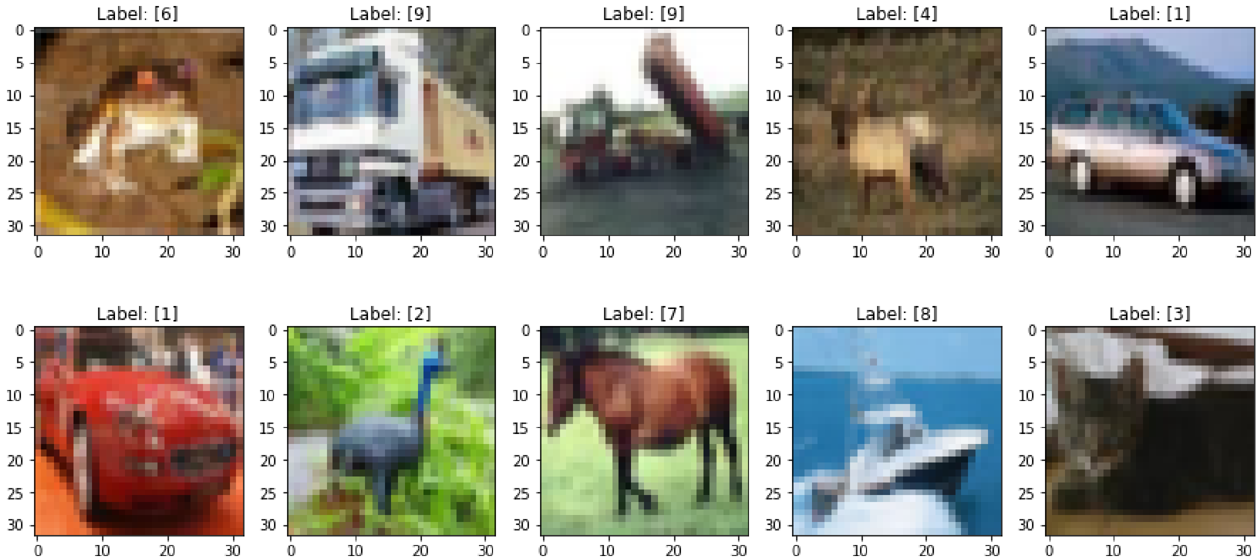
```
import matplotlib.pyplot as plt
%matplotlib inline

# specify the number of rows and columns you want to see
num_row = 2
num_col = 5

# get a segment of the dataset
num = num_row*num_col
images = x_train[:num]
labels = y_train[:num]

# plot images
fig, axes = plt.subplots(num_row, num_col, figsize=(2.5*num_col,3*num_row))
for i in range(num_row*num_col):
    ax = axes[i//num_col, i%num_col]
    ax.imshow(images[i])
    ax.set_title('Label: {}'.format(labels[i]))
```

```
plt.tight_layout()
plt.show()
```



5. Réalisez une standardisation des deux sous-ensembles des données (normalisation des valeurs des pixels)

In [163…

```
z_train = x_train.astype('float32') / 255
z_test = x_test.astype('float32') / 255
```

In [164…

```
#one hot encoding pour y, pour utiliser categorical_crossentropy pour la multiclassific

from keras.utils import np_utils
ymTrain = np_utils.to_categorical(y_train)
ymTest = np_utils.to_categorical(y_test)
```

In [165…

```
# Verification
ymTest
```

Out[165…

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 1., 0., 0.]], dtype=float32)
```

In [166…

```
ymTest.shape  # Verification de 10 columns, une pour chaque class
```

Out[166…

```
(10000, 10)
```

6. Construisez une réseau de neurone convolutif (CNN) ayant l’architecture suivante :

- Utilisez trois filtres (couches de convolution) de dimension 3 qui progressent selon les couches selon : 32, puis 64, puis 128.
- Utilisez une normalisation par Batch après chaque couche de convolution et un maxPooloing de taille 2
- Utilisez la fonction d'activation relu et un padding='same'.
- Utilisez ensuite le réseau intégralement connecté, constitué d'une couche dense de taille 1024
- Utilisez également un dropout avec un taux d'extinction de 20%

In [167…

```
# Create model
model = tf.keras.models.Sequential()

# 3 Couches de convolution et normalisation par Batch après chaque couche de convolutio
model.add(tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape
tf.keras.layers.BatchNormalization(),
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2))),

model.add(tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation='relu', padding="SA
tf.keras.layers.BatchNormalization(),
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2))),

model.add(tf.keras.layers.Conv2D(128, kernel_size=(3, 3), activation='relu', padding="S
tf.keras.layers.BatchNormalization(),
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2))),

# réseau intégralement connecté, constitué d'une couche dense de taille 1024
model.add(tf.keras.layers.Flatten()),
model.add(tf.keras.layers.Dense(1024, activation='relu')),
model.add(tf.keras.layers.Dropout(0.2)),
model.add(tf.keras.layers.Dense(10, activation='softmax'))
```

7. Affichez le sommaire du modèle

In [168…

```
# Sommaire du modèle B
model.summary()
```

Model: "sequential_8"

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d_9 (MaxPooling2	(None, 16, 16, 32)	0
conv2d_10 (Conv2D)	(None, 16, 16, 64)	18496
max_pooling2d_10 (MaxPooling	(None, 8, 8, 64)	0
conv2d_11 (Conv2D)	(None, 8, 8, 128)	73856
max_pooling2d_11 (MaxPooling	(None, 4, 4, 128)	0
flatten_7 (Flatten)	(None, 2048)	0
dense_50 (Dense)	(None, 1024)	2098176
dropout_3 (Dropout)	(None, 1024)	0

```
_________________________________________________________________
dense_51 (Dense)             (None, 10)                10250     
=================================================================
Total params: 2,201,674
Trainable params: 2,201,674
Non-trainable params: 0
_________________________________________________________________
```

8. En utilisant l'optimisation adam, entrainez le CNN sur le jeu de données d'entrainement de CIFAR10 pendant 50 époques. La métrique utilisée étant la valeur de l'accuracy.

In [169…

```python
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit(z_train, ymTrain, shuffle=True, epochs=50, validation_data=(z_test,
```

```
Epoch 1/50
1563/1563 [==============================] - 125s 76ms/step - loss: 1.3233 - accuracy:
0.5230 - val_loss: 0.9983 - val_accuracy: 0.6474
Epoch 2/50
1563/1563 [==============================] - 106s 68ms/step - loss: 0.9073 - accuracy:
0.6789 - val_loss: 0.8586 - val_accuracy: 0.6996
Epoch 3/50
1563/1563 [==============================] - 119s 76ms/step - loss: 0.7358 - accuracy:
0.7416 - val_loss: 0.7857 - val_accuracy: 0.7285
Epoch 4/50
1563/1563 [==============================] - 117s 75ms/step - loss: 0.6092 - accuracy:
0.7867 - val_loss: 0.7426 - val_accuracy: 0.7511
Epoch 5/50
1563/1563 [==============================] - 120s 77ms/step - loss: 0.4935 - accuracy:
0.8249 - val_loss: 0.7931 - val_accuracy: 0.7404
Epoch 6/50
1563/1563 [==============================] - 115s 73ms/step - loss: 0.3952 - accuracy:
0.8618 - val_loss: 0.7741 - val_accuracy: 0.7567
Epoch 7/50
1563/1563 [==============================] - 107s 68ms/step - loss: 0.3113 - accuracy:
0.8912 - val_loss: 0.8418 - val_accuracy: 0.7536
Epoch 8/50
1563/1563 [==============================] - 123s 78ms/step - loss: 0.2520 - accuracy:
0.9115 - val_loss: 0.9277 - val_accuracy: 0.7463
Epoch 9/50
1563/1563 [==============================] - 114s 73ms/step - loss: 0.2007 - accuracy:
0.9294 - val_loss: 0.9471 - val_accuracy: 0.7587
Epoch 10/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.1750 - accuracy:
0.9387 - val_loss: 1.0442 - val_accuracy: 0.7561
Epoch 11/50
1563/1563 [==============================] - 109s 69ms/step - loss: 0.1514 - accuracy:
0.9475 - val_loss: 1.1729 - val_accuracy: 0.7501
Epoch 12/50
1563/1563 [==============================] - 118s 76ms/step - loss: 0.1401 - accuracy:
0.9522 - val_loss: 1.1543 - val_accuracy: 0.7538
Epoch 13/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.1351 - accuracy:
0.9539 - val_loss: 1.2462 - val_accuracy: 0.7506
Epoch 14/50
1563/1563 [==============================] - 121s 78ms/step - loss: 0.1198 - accuracy:
0.9596 - val_loss: 1.3410 - val_accuracy: 0.7580
Epoch 15/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.1231 - accuracy:
0.9590 - val_loss: 1.3801 - val_accuracy: 0.7498
Epoch 16/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.1118 - accuracy:
```

```
0.9618 - val_loss: 1.4224 - val_accuracy: 0.7555
Epoch 17/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.1097 - accuracy:
0.9633 - val_loss: 1.5067 - val_accuracy: 0.7526
Epoch 18/50
1563/1563 [==============================] - 121s 78ms/step - loss: 0.1075 - accuracy:
0.9646 - val_loss: 1.4899 - val_accuracy: 0.7519
Epoch 19/50
1563/1563 [==============================] - 111s 71ms/step - loss: 0.0981 - accuracy:
0.9677 - val_loss: 1.4860 - val_accuracy: 0.7558
Epoch 20/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.1043 - accuracy:
0.9667 - val_loss: 1.5643 - val_accuracy: 0.7398
Epoch 21/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0963 - accuracy:
0.9692 - val_loss: 1.6656 - val_accuracy: 0.7545
Epoch 22/50
1563/1563 [==============================] - 112s 71ms/step - loss: 0.0944 - accuracy:
0.9700 - val_loss: 1.6902 - val_accuracy: 0.7440
Epoch 23/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0885 - accuracy:
0.9726 - val_loss: 1.6891 - val_accuracy: 0.7505
Epoch 24/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0926 - accuracy:
0.9715 - val_loss: 1.8103 - val_accuracy: 0.7517
Epoch 25/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0933 - accuracy:
0.9699 - val_loss: 1.7711 - val_accuracy: 0.7517
Epoch 26/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.1021 - accuracy:
0.9690 - val_loss: 1.8200 - val_accuracy: 0.7484
Epoch 27/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0792 - accuracy:
0.9752 - val_loss: 1.8534 - val_accuracy: 0.7479
Epoch 28/50
1563/1563 [==============================] - 111s 71ms/step - loss: 0.0823 - accuracy:
0.9755 - val_loss: 1.9186 - val_accuracy: 0.7501
Epoch 29/50
1563/1563 [==============================] - 111s 71ms/step - loss: 0.0905 - accuracy:
0.9730 - val_loss: 1.8699 - val_accuracy: 0.7467
Epoch 30/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0903 - accuracy:
0.9741 - val_loss: 1.9118 - val_accuracy: 0.7475
Epoch 31/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0870 - accuracy:
0.9749 - val_loss: 2.0244 - val_accuracy: 0.7498
Epoch 32/50
1563/1563 [==============================] - 110s 71ms/step - loss: 0.0918 - accuracy:
0.9733 - val_loss: 2.0442 - val_accuracy: 0.7398
Epoch 33/50
1563/1563 [==============================] - 111s 71ms/step - loss: 0.0864 - accuracy:
0.9750 - val_loss: 2.1408 - val_accuracy: 0.7448
Epoch 34/50
1563/1563 [==============================] - 111s 71ms/step - loss: 0.0857 - accuracy:
0.9745 - val_loss: 2.0345 - val_accuracy: 0.7502
Epoch 35/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0871 - accuracy:
0.9766 - val_loss: 2.2032 - val_accuracy: 0.7435
Epoch 36/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.0804 - accuracy:
0.9769 - val_loss: 2.3901 - val_accuracy: 0.7335
Epoch 37/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.0868 - accuracy:
0.9756 - val_loss: 2.0964 - val_accuracy: 0.7504
Epoch 38/50
```

```
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0846 - accuracy: 0.9760 - val_loss: 2.1849 - val_accuracy: 0.7511
Epoch 39/50
1563/1563 [==============================] - 109s 69ms/step - loss: 0.0842 - accuracy: 0.9772 - val_loss: 2.1492 - val_accuracy: 0.7448
Epoch 40/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0835 - accuracy: 0.9770 - val_loss: 2.2539 - val_accuracy: 0.7507
Epoch 41/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0727 - accuracy: 0.9796 - val_loss: 2.4493 - val_accuracy: 0.7479
Epoch 42/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0776 - accuracy: 0.9787 - val_loss: 2.4059 - val_accuracy: 0.7507
Epoch 43/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0861 - accuracy: 0.9770 - val_loss: 2.3373 - val_accuracy: 0.7446
Epoch 44/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0792 - accuracy: 0.9795 - val_loss: 2.4928 - val_accuracy: 0.7449
Epoch 45/50
1563/1563 [==============================] - 110s 70ms/step - loss: 0.0881 - accuracy: 0.9769 - val_loss: 2.5328 - val_accuracy: 0.7403
Epoch 46/50
1563/1563 [==============================] - 109s 70ms/step - loss: 0.0820 - accuracy: 0.9787 - val_loss: 2.6056 - val_accuracy: 0.7470
Epoch 47/50
1563/1563 [==============================] - 110s 71ms/step - loss: 0.0881 - accuracy: 0.9763 - val_loss: 2.6836 - val_accuracy: 0.7472
Epoch 48/50
1563/1563 [==============================] - 125s 80ms/step - loss: 0.0814 - accuracy: 0.9796 - val_loss: 2.5301 - val_accuracy: 0.7355
Epoch 49/50
1563/1563 [==============================] - 108s 69ms/step - loss: 0.0844 - accuracy: 0.9785 - val_loss: 2.8098 - val_accuracy: 0.7304
Epoch 50/50
1563/1563 [==============================] - 110s 71ms/step - loss: 0.0806 - accuracy: 0.9791 - val_loss: 2.8401 - val_accuracy: 0.7370
```

In [ ]:

9. Représentez la matrice de confusion sur les données de test. Commentez les résultats

9.1. Prediction

In [180…

```
# Prédiction sur l'ensemble des données de test
yPred = model.predict(z_test)
print(yPred.shape)
```

```
(10000, 10)
```

In [181…

```
yPred
```

Out[181…

```
array([[3.88e-14, 1.34e-14, 9.35e-15, ..., 1.14e-15, 1.09e-08, 7.11e-19],
       [9.19e-13, 9.97e-01, 4.84e-37, ..., 0.00e+00, 3.01e-03, 3.21e-21],
       [8.77e-02, 1.57e-07, 2.37e-12, ..., 5.67e-11, 9.12e-01, 1.56e-04],
       ...,
       [0.00e+00, 0.00e+00, 5.15e-28, ..., 0.00e+00, 0.00e+00, 0.00e+00],
```

```
        [2.21e-16, 1.00e+00, 2.42e-23, ..., 2.25e-26, 8.30e-23, 1.89e-28],
        [0.00e+00, 0.00e+00, 0.00e+00, ..., 1.00e+00, 0.00e+00, 0.00e+00]],
       dtype=float32)
```

In [182…

```
import numpy

iYPred = numpy.argmax(yPred,axis=1)
iYPred
```

Out[182…

```
array([3, 1, 8, ..., 5, 1, 7], dtype=int64)
```

9.2. Verification de l'approche et de resultat pour une image.

In [183…

```
iYPred.shape
```

Out[183…

```
(10000,)
```

In [194…

```
# Verification 3e image de set

# 0: airplane
# 1: automobile
# 2: bird
# 3: cat
# 4: deer
# 5: dog
# 6: frog
# 7: horse
# 8: ship
# 9: truck

# Affichage de la première ligne, la prediction correspondent cest la class 8 (ship)
# Le valeur max cest 1 dans la position de la class 8

print(yPred[2,:]) #---> Class 8 --> ship
```

```
[8.77e-02 1.57e-07 2.37e-12 5.00e-13 3.38e-14 2.47e-13 3.78e-10 5.67e-11
 9.12e-01 1.56e-04]
```

In [195…

```
iYPredx = numpy.argmax(yPred[2,:])
iYPredx
```

Out[195…

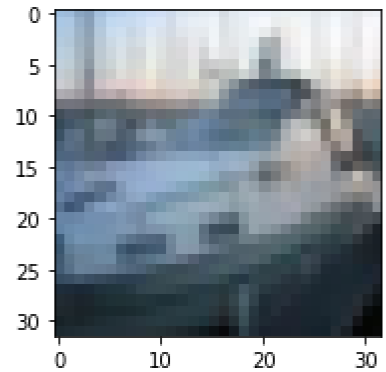
```
8
```

In [202…

```
# pick le image 3
sample = 2
image = x_test[sample]

fig = plt.figure(figsize = (3,3))
plt.imshow(image)
plt.show()

## oh yeah :)  La prediction est correct.
```



9.3. Matrix de confusion

In [186…

```
from sklearn.metrics import classification_report,confusion_matrix
cnf_matrix = confusion_matrix(y_true=y_test, y_pred=iYPred)
np.set_printoptions(precision=2)
cnf_matrix
```

Out[186…

```
array([[806,  17,  53,  10,  17,  10,  13,  11,  41,  22],
       [ 14, 877,   6,   6,   5,   1,  10,   3,  14,  64],
       [ 58,   7, 677,  34,  52,  62,  66,  23,  13,   8],
       [ 23,  14,  78, 500,  51, 157, 117,  31,  14,  15],
       [ 19,   1, 113,  51, 648,  43,  64,  55,   4,   2],
       [ 18,   3,  52, 126,  41, 644,  60,  40,   6,  10],
       [  3,   6,  51,  42,  17,  14, 854,   3,   5,   5],
       [ 15,   2,  39,  34,  49,  57,  15, 771,   3,  15],
       [ 78,  45,  10,  15,   7,   7,  10,   3, 790,  35],
       [ 28,  97,  14,  13,   6,   7,   4,  10,  18, 803]], dtype=int64)
```

9.4. Matrix de confusion normalize

Pour faciliter l'interpretation de resultats

In [187…

```
cnf_matrixN = confusion_matrix(y_true=y_test, y_pred=iYPred, normalize='true')
np.set_printoptions(precision=2)
cnf_matrixN
```

Out[187…

```
array([[0.81, 0.02, 0.05, 0.01, 0.02, 0.01, 0.01, 0.01, 0.04, 0.02],
       [0.01, 0.88, 0.01, 0.01, 0.01, 0.  , 0.01, 0.  , 0.01, 0.06],
       [0.06, 0.01, 0.68, 0.03, 0.05, 0.06, 0.07, 0.02, 0.01, 0.01],
       [0.02, 0.01, 0.08, 0.5 , 0.05, 0.16, 0.12, 0.03, 0.01, 0.01],
       [0.02, 0.  , 0.11, 0.05, 0.65, 0.04, 0.06, 0.06, 0.  , 0.  ],
       [0.02, 0.  , 0.05, 0.13, 0.04, 0.64, 0.06, 0.04, 0.01, 0.01],
       [0.  , 0.01, 0.05, 0.04, 0.02, 0.01, 0.85, 0.  , 0.01, 0.01],
       [0.01, 0.  , 0.04, 0.03, 0.05, 0.06, 0.01, 0.77, 0.  , 0.01],
       [0.08, 0.04, 0.01, 0.01, 0.01, 0.01, 0.01, 0.  , 0.79, 0.04],
       [0.03, 0.1 , 0.01, 0.01, 0.01, 0.01, 0.  , 0.01, 0.02, 0.8 ]])
```

In [188…

```
# Analysis:

# Diagonal represents le taux de classification pour chaque class, la class que mieux a
# la class 1 (Autombile), avec 88% correctement classifie. Par contre la class avec plu
# la class 3 (cat), avec 50% correctement classifie.

# On observe que ilya confusion dans l<algorithme pour classifier le chats, l<algorithm
```

```
# chats comme class 5 (dogs), la meme observation on peut faire, que pour les chiennes
# classification correct de 64% et ilya confusion avec les chats de 16%.
```

9.4. Classification_report

Pour faciliter la comparation des modeles

In [217…

```
print(classification_report(y_true=y_test, y_pred=iYPred))
```

```
              precision    recall  f1-score   support

           0       0.76      0.81      0.78      1000
           1       0.82      0.88      0.85      1000
           2       0.62      0.68      0.65      1000
           3       0.60      0.50      0.55      1000
           4       0.73      0.65      0.68      1000
           5       0.64      0.64      0.64      1000
           6       0.70      0.85      0.77      1000
           7       0.81      0.77      0.79      1000
           8       0.87      0.79      0.83      1000
           9       0.82      0.80      0.81      1000

    accuracy                           0.74     10000
   macro avg       0.74      0.74      0.74     10000
weighted avg       0.74      0.74      0.74     10000
```

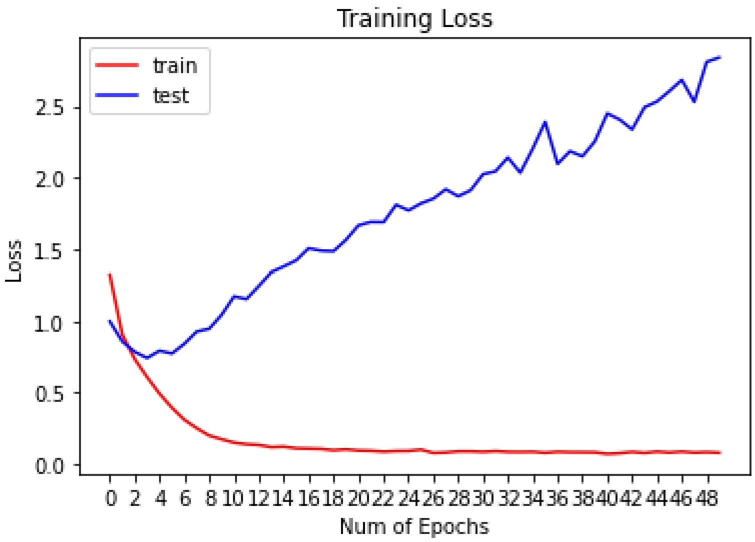
10. Représentez la courbe de variation de la fonction perte sur les données d'entrainement et de test.

In [190…

```
plt.figure(1)
plt.plot(history.history['loss'],'r')
plt.plot(history.history['val_loss'],'b')
plt.xticks(np.arange(0, 50, 2.0))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Loss")
plt.title("Training Loss")
plt.legend(['train', 'test'])
```

Out[190…

```
<matplotlib.legend.Legend at 0x175998a4bb0>
```



In [ ]:

```
# De la courve d'Apprentissage on peut deduire une overfitting, ou le modele est capabl
# les donnes de entrenainment mais il est pas capable de le faire pour les nouvelles do
# cest une situation indeserable et plus critique parce que le point d inflection est d
```

11. Construisez un RNP dense constitué de 10 couches cachées, chacune avec 100 neurones. Utilisez l'initialisation de He et la fonction d'activation ELU

In [191…

```
model_B = tf.keras.models.Sequential()
model_B.add(tf.keras.layers.Flatten(input_shape=[32, 32, 3]))
for x in range(10):
    model_B.add(tf.keras.layers.Dense(100, kernel_initializer="he_normal"))
    model_B.add(tf.keras.layers.Activation("elu"))

model_B.add(tf.keras.layers.Dense(10, activation="softmax"))
```

In [ ]:

```

```

12. En utilisant l'optimisation Nadam (learning_rate=5e-5), entrainez le RNP sur le jeu de données d'entrainement CIFAR10 pendant 50 époques. La métrique utilisée étant l'accuracy

In [192…

```
optimizer = tf.keras.optimizers.Nadam(learning_rate=5e-5)
model_B.compile(loss="categorical_crossentropy", optimizer=optimizer, metrics=["accurac
history2 = model_B.fit(z_train, ymTrain, epochs=50, validation_data=(z_test, ymTest))
```

```
Epoch 1/50
1563/1563 [==============================] - 29s 13ms/step - loss: 1.8692 - accuracy: 0.
3265 - val_loss: 1.7231 - val_accuracy: 0.3793
Epoch 2/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.6635 - accuracy: 0.
4044 - val_loss: 1.6146 - val_accuracy: 0.4220
Epoch 3/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.5760 - accuracy: 0.
4378 - val_loss: 1.5625 - val_accuracy: 0.4439
Epoch 4/50
```

```
1563/1563 [==============================] - 18s 11ms/step - loss: 1.5210 - accuracy: 0.
4579 - val_loss: 1.5230 - val_accuracy: 0.4590
Epoch 5/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.4799 - accuracy: 0.
4719 - val_loss: 1.5183 - val_accuracy: 0.4603
Epoch 6/50
1563/1563 [==============================] - 20s 13ms/step - loss: 1.4466 - accuracy: 0.
4822 - val_loss: 1.4980 - val_accuracy: 0.4712
Epoch 7/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.4183 - accuracy: 0.
4934 - val_loss: 1.4607 - val_accuracy: 0.4777
Epoch 8/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.3932 - accuracy: 0.
5012 - val_loss: 1.4835 - val_accuracy: 0.4712
Epoch 9/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.3688 - accuracy: 0.
5109 - val_loss: 1.4736 - val_accuracy: 0.4756
Epoch 10/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.3491 - accuracy: 0.
5189 - val_loss: 1.4701 - val_accuracy: 0.4736
Epoch 11/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.3279 - accuracy: 0.
5271 - val_loss: 1.4416 - val_accuracy: 0.4876
Epoch 12/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.3110 - accuracy: 0.
5318 - val_loss: 1.4498 - val_accuracy: 0.4889
Epoch 13/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.2927 - accuracy: 0.
5376 - val_loss: 1.3929 - val_accuracy: 0.5091
Epoch 14/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.2758 - accuracy: 0.
5442 - val_loss: 1.4071 - val_accuracy: 0.5070
Epoch 15/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.2584 - accuracy: 0.
5495 - val_loss: 1.4235 - val_accuracy: 0.4942
Epoch 16/50
1563/1563 [==============================] - 30s 19ms/step - loss: 1.2438 - accuracy: 0.
5554 - val_loss: 1.4056 - val_accuracy: 0.5048
Epoch 17/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.2297 - accuracy: 0.
5594 - val_loss: 1.3851 - val_accuracy: 0.5138
Epoch 18/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.2150 - accuracy: 0.
5649 - val_loss: 1.3895 - val_accuracy: 0.5082
Epoch 19/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.2007 - accuracy: 0.
5704 - val_loss: 1.4010 - val_accuracy: 0.5126
Epoch 20/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.1874 - accuracy: 0.
5747 - val_loss: 1.3923 - val_accuracy: 0.5140
Epoch 21/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.1726 - accuracy: 0.
5799 - val_loss: 1.4069 - val_accuracy: 0.5045
Epoch 22/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.1609 - accuracy: 0.
5824 - val_loss: 1.3944 - val_accuracy: 0.5108
Epoch 23/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.1478 - accuracy: 0.
5899 - val_loss: 1.4039 - val_accuracy: 0.5124
Epoch 24/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.1368 - accuracy: 0.
5922 - val_loss: 1.4115 - val_accuracy: 0.5147
Epoch 25/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.1254 - accuracy: 0.
5984 - val_loss: 1.3928 - val_accuracy: 0.5183
```

```
Epoch 26/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.1134 - accuracy: 0.
6018 - val_loss: 1.3916 - val_accuracy: 0.5154
Epoch 27/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.1034 - accuracy: 0.
6058 - val_loss: 1.4135 - val_accuracy: 0.5187
Epoch 28/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.0939 - accuracy: 0.
6112 - val_loss: 1.4122 - val_accuracy: 0.5207
Epoch 29/50
1563/1563 [==============================] - 17s 11ms/step - loss: 1.0806 - accuracy: 0.
6139 - val_loss: 1.4209 - val_accuracy: 0.5104
Epoch 30/50
1563/1563 [==============================] - 19s 12ms/step - loss: 1.0709 - accuracy: 0.
6167 - val_loss: 1.4053 - val_accuracy: 0.5195
Epoch 31/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.0575 - accuracy: 0.
6227 - val_loss: 1.4107 - val_accuracy: 0.5211
Epoch 32/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.0486 - accuracy: 0.
6239 - val_loss: 1.4055 - val_accuracy: 0.5212
Epoch 33/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.0411 - accuracy: 0.
6293 - val_loss: 1.5127 - val_accuracy: 0.4970
Epoch 34/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.0279 - accuracy: 0.
6354 - val_loss: 1.4313 - val_accuracy: 0.5063
Epoch 35/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.0173 - accuracy: 0.
6345 - val_loss: 1.4590 - val_accuracy: 0.5107
Epoch 36/50
1563/1563 [==============================] - 18s 12ms/step - loss: 1.0083 - accuracy: 0.
6407 - val_loss: 1.4348 - val_accuracy: 0.5161
Epoch 37/50
1563/1563 [==============================] - 18s 11ms/step - loss: 1.0015 - accuracy: 0.
6418 - val_loss: 1.4253 - val_accuracy: 0.5164
Epoch 38/50
1563/1563 [==============================] - 17s 11ms/step - loss: 0.9901 - accuracy: 0.
6484 - val_loss: 1.4275 - val_accuracy: 0.5213
Epoch 39/50
1563/1563 [==============================] - 17s 11ms/step - loss: 0.9793 - accuracy: 0.
6502 - val_loss: 1.4369 - val_accuracy: 0.5203
Epoch 40/50
1563/1563 [==============================] - 19s 12ms/step - loss: 0.9710 - accuracy: 0.
6556 - val_loss: 1.4446 - val_accuracy: 0.5256
Epoch 41/50
1563/1563 [==============================] - 17s 11ms/step - loss: 0.9617 - accuracy: 0.
6580 - val_loss: 1.4547 - val_accuracy: 0.5226
Epoch 42/50
1563/1563 [==============================] - 17s 11ms/step - loss: 0.9516 - accuracy: 0.
6608 - val_loss: 1.4438 - val_accuracy: 0.5209
Epoch 43/50
1563/1563 [==============================] - 18s 11ms/step - loss: 0.9425 - accuracy: 0.
6632 - val_loss: 1.4636 - val_accuracy: 0.5197
Epoch 44/50
1563/1563 [==============================] - 17s 11ms/step - loss: 0.9317 - accuracy: 0.
6684 - val_loss: 1.4460 - val_accuracy: 0.5257
Epoch 45/50
1563/1563 [==============================] - 18s 11ms/step - loss: 0.9251 - accuracy: 0.
6704 - val_loss: 1.4700 - val_accuracy: 0.5272
Epoch 46/50
1563/1563 [==============================] - 18s 11ms/step - loss: 0.9168 - accuracy: 0.
6734 - val_loss: 1.4775 - val_accuracy: 0.5196
Epoch 47/50
1563/1563 [==============================] - 20s 13ms/step - loss: 0.9082 - accuracy: 0.
```

```
6770 - val_loss: 1.5139 - val_accuracy: 0.5181
Epoch 48/50
1563/1563 [==============================] - 20s 13ms/step - loss: 0.8982 - accuracy: 0.
6821 - val_loss: 1.4911 - val_accuracy: 0.5207
Epoch 49/50
1563/1563 [==============================] - 18s 11ms/step - loss: 0.8887 - accuracy: 0.
6846 - val_loss: 1.4900 - val_accuracy: 0.5224
Epoch 50/50
1563/1563 [==============================] - 18s 11ms/step - loss: 0.8831 - accuracy: 0.
6866 - val_loss: 1.5121 - val_accuracy: 0.5214
```

In [ ]:

13. Représentez la matrice de confusion. Commentez les résultats.

13.1. Prediction

In [203…

```
# Prédiction sur l'ensemble des données de test
yPredB = model_B.predict(z_test)
print(yPredB.shape)
```

```
(10000, 10)
```

In [204…

```
import numpy

iYPredB = numpy.argmax(yPredB,axis=1)
iYPredB
```

Out[204…

```
array([3, 8, 8, ..., 2, 4, 7], dtype=int64)
```

13.2. Verification de l'approche et de resultat pour une image.

In [205…

```
iYPredB.shape
```

Out[205…

```
(10000,)
```

In [209…

```
# Verification 3e image de set

# 0: airplane
# 1: automobile
# 2: bird
# 3: cat
# 4: deer
# 5: dog
# 6: frog
# 7: horse
# 8: ship
# 9: truck

# Affichage de la première ligne, la prediction correspondent cest la class 8 (ship)
# Le valeur max cest 1 dans la position de la class 8

print(yPredB[2,:]) #---> Class 8 --> ship
```

```
[4.36e-01 1.44e-02 2.10e-03 3.45e-04 2.98e-04 7.59e-04 1.41e-05 2.64e-04
 5.17e-01 2.82e-02]
```

In [210…

```
iYPredx = numpy.argmax(yPredB[1,:])
iYPredx
```

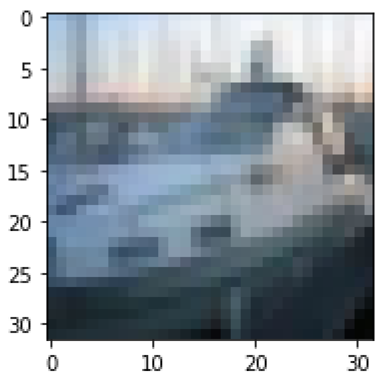
Out[210… 8

In [212…

```
# pick le 3e image
sample = 2
image = x_test[sample]

fig = plt.figure(figsize = (3,3))
plt.imshow(image)
plt.show()

## oh yeah :)  La prediction est correct.
```



13.3. Matrix de confusion

In [218…

```
from sklearn.metrics import classification_report,confusion_matrix
cnf_matrixB = confusion_matrix(y_true=y_test, y_pred=iYPredB)
np.set_printoptions(precision=2)
cnf_matrixB
```

Out[218…

```
array([[615,  21,  56,  19,  30,  19,  21,  19, 161,  39],
       [ 65, 572,  19,  25,  12,  17,  16,  18, 107, 149],
       [ 83,  14, 392,  82, 152,  91,  76,  62,  30,  18],
       [ 39,  21,  91, 318,  86, 245,  90,  41,  36,  33],
       [ 58,  12, 121,  53, 476,  68,  88,  76,  36,  12],
       [ 23,   8,  88, 175,  85, 458,  52,  51,  32,  28],
       [  7,  15,  93,  83, 110,  79, 566,  13,  19,  15],
       [ 46,  11,  66,  54,  93,  88,  18, 546,  27,  51],
       [ 87,  45,   9,  21,  17,  25,   9,   9, 744,  34],
       [ 52, 142,  16,  35,   9,  22,  24,  42, 131, 527]], dtype=int64)
```

13.4. Matrix de confusion normalize

Pour faciliter l'interpretation de resultats

In [219…

```
cnf_matrixNB = confusion_matrix(y_true=y_test, y_pred=iYPredB, normalize='true')
np.set_printoptions(precision=2)
cnf_matrixNB
```

Out[219…
```
array([[0.61, 0.02, 0.06, 0.02, 0.03, 0.02, 0.02, 0.02, 0.16, 0.04],
       [0.07, 0.57, 0.02, 0.03, 0.01, 0.02, 0.02, 0.02, 0.11, 0.15],
       [0.08, 0.01, 0.39, 0.08, 0.15, 0.09, 0.08, 0.06, 0.03, 0.02],
       [0.04, 0.02, 0.09, 0.32, 0.09, 0.24, 0.09, 0.04, 0.04, 0.03],
       [0.06, 0.01, 0.12, 0.05, 0.48, 0.07, 0.09, 0.08, 0.04, 0.01],
       [0.02, 0.01, 0.09, 0.17, 0.09, 0.46, 0.05, 0.05, 0.03, 0.03],
       [0.01, 0.01, 0.09, 0.08, 0.11, 0.08, 0.57, 0.01, 0.02, 0.01],
       [0.05, 0.01, 0.07, 0.05, 0.09, 0.09, 0.02, 0.55, 0.03, 0.05],
       [0.09, 0.04, 0.01, 0.02, 0.02, 0.03, 0.01, 0.01, 0.74, 0.03],
       [0.05, 0.14, 0.02, 0.04, 0.01, 0.02, 0.02, 0.04, 0.13, 0.53]])
```

In [ ]:
```
# Analysis:

# Diagonal represents le taux de classification pour chaque class, la class que mieux a
# la class 8 (Ship), avec 74% correctement classifie. Par contre la class avec plus des
# la class 3 (cat), avec 32% correctement classifie.

# On observe que ilya confusion dans l'algorithme pour classifier les chats, l'algorith
# chats comme class 5 (dogs), la meme observation on peut faire, que pour les chiennes
# classification correct de 46% et ilya confusion avec les chats de 24%.
```

13.4. Classification_report

Pour faciliter la comparation des modeles

In [215…
```
print(classification_report(y_true=y_test, y_pred=iYPredB))
```

```
              precision    recall  f1-score   support

           0       0.57      0.61      0.59      1000
           1       0.66      0.57      0.61      1000
           2       0.41      0.39      0.40      1000
           3       0.37      0.32      0.34      1000
           4       0.44      0.48      0.46      1000
           5       0.41      0.46      0.43      1000
           6       0.59      0.57      0.58      1000
           7       0.62      0.55      0.58      1000
           8       0.56      0.74      0.64      1000
           9       0.58      0.53      0.55      1000

    accuracy                           0.52     10000
   macro avg       0.52      0.52      0.52     10000
weighted avg       0.52      0.52      0.52     10000
```

In [216…
```
plt.figure(1)
plt.plot(history2.history['loss'],'r')
plt.plot(history2.history['val_loss'],'b')
plt.xticks(np.arange(0, 50, 2.0))
plt.rcParams['figure.figsize'] = (8, 6)
plt.xlabel("Num of Epochs")
plt.ylabel("Loss")
plt.title("Training Loss")
plt.legend(['train', 'test'])
```

Out[216…
```
<matplotlib.legend.Legend at 0x175b019be80>
```

In [220…

```
# De la courve d'Apprentissage on peut deduire une overfitting, ou le modele est capabl
# les donnes de entrenainment mais il est pas capable de le faire pour les nouvelles do
# cest une situation indeserable et plus critique parce que le point d inflection est d

# Aussi on peut observer que on a arrete le entrainment de facon premature, avec plus d
# le comportement de set de entrainement.
```

14. Comparez les deux modèles CNN et RNP obtenus et commentez les résultats.

In [226…

```
# Pour le modele CNN on a uen taux de classification globale de 74% et pour RNP 52%
# Les deux models ont comportement de overfiting, sont capables de bien classifier les
# de training (CNN mieux que RNP),
# mais ne sont pas bonnes pour classifier les nouvelles donnes.
# Les deux modeles present de confusion entre les chats et les chiennes.
```

In [227…

```
test_eval = model.evaluate(z_test, ymTest, verbose=0)
print('Test loss:', test_eval[0])
print('Test accuracy:', test_eval[1])
```

```
Test loss: 2.8400843143463135
Test accuracy: 0.7369999885559082
```

In [228…

```
test_evalB = model_B.evaluate(z_test, ymTest, verbose=0)
print('Test loss:', test_evalB[0])
print('Test accuracy:', test_evalB[1])
```

```
Test loss: 1.5120548009872437
Test accuracy: 0.521399974822998
```