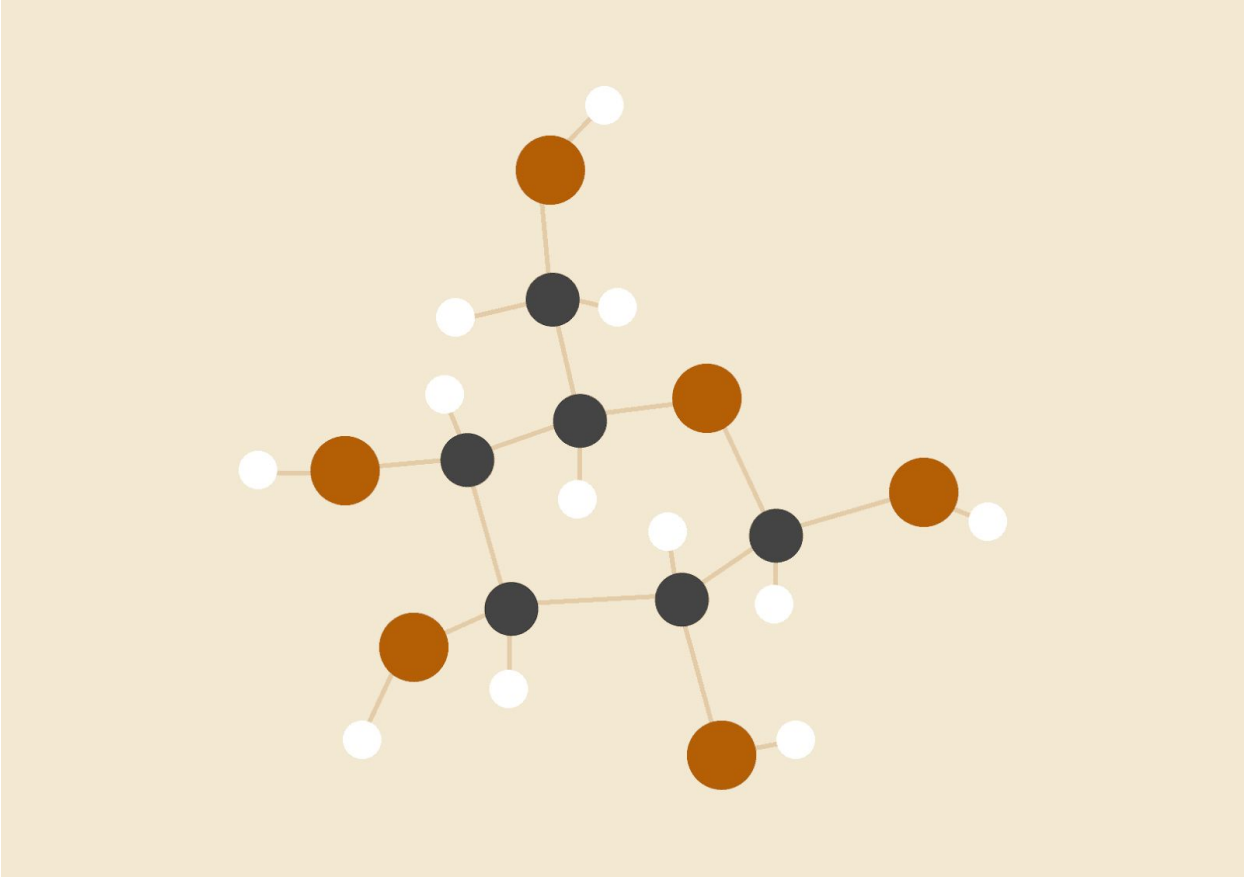
RAPPORT TP2 APPRENTISSAGE AUTOMATIQUE



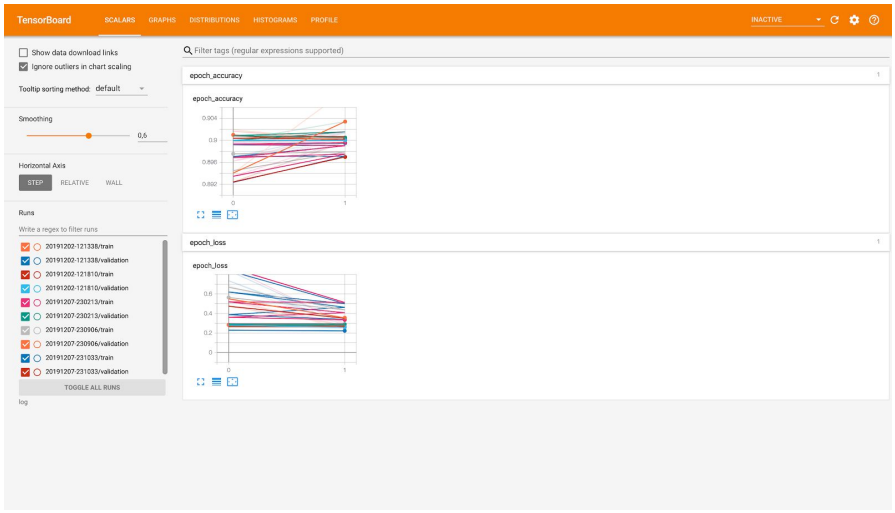
BOURRELY Thomas

08/12/2019

Question 1

Les options suivantes permettent d'afficher les histogrammes des gradients des différentes couches.

```
histogram_freq=1
write_grads=True
```



Question 2

Afin de sauvegarder la meilleure version, j'utilise le callback suivant.

```
model_save_filepath = "models/best-model.hdf5"

modelCheckpointCallback = ModelCheckpoint(
    model_save_filepath,
    monitor='val_accuracy',
    verbose=1,
    save_best_only=True,
    save_weights_only=False,
    mode='max',
    period=1
)
```

Le modèle est ensuite chargé dans le script "testing.py".

```
# from ...

# Process mnist dataset and return only test data
def mnistTestData():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    input_dimension = 784
    num_classes = 10

    x_test = x_test.reshape(10000, input_dimension)
    x_test = x_test.astype('float32')
    x_test /= 255

    y_test = np_utils.to_categorical(y_test, num_classes)

    return (x_test, y_test)

# Evaluate a model and print the score data
def evaluateModel(model, x_test, y_test, verbose=0):
    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test score: {}'.format(score[0]))
    print('Test accuracy: {}'.format(score[1]))
```

```
# Test if file exists, if not -> exit
def quitIfFileNotExist(file):
    if not path.exists(file):
        print(file + ' does not exists')
        sys.exit(1)

#==================================================================

bestModelPath = 'models/best-model.hdf5'

quitIfFileNotExist(bestModelPath)
model = load_model(bestModelPath)
(x_test, y_test) = mnistTestData()
evaluateModel(model, x_test, y_test)
```

Question 3

Préparation des données

Chargement des données avec pandas :

```
completeDataset = pandas.read_csv(
   dataSource,
   sep='\t',
   usecols=['voyelle', 'F1', 'F2', 'F3', 'F4', 'Z1', 'Z2', 'f0']
)
```

Suppression des lignes “NaN” et “--undefined--” :

```
completeDataset.dropna()
completeDataset = completeDataset[completeDataset['F4'] != '--undefined--']
```

Passage de toutes les données en type “float” :

```
completeDataset[['F1', 'F2', 'F3', 'F4', 'f0']] = completeDataset[['F1', 'F2', 'F3', 'F4',
'f0']].astype(float)
```

Séparation des données en 80% training, 10% validation et 10% tests :

```
splittingSourceDataset = completeDataset.copy()

# Use 80% as training
training_sample = splittingSourceDataset.sample(frac=0.8, random_state=0)
x_train, y_train = dataframeToXandY(training_sample)

# Update splitting source dataset
splittingSourceDataset = splittingSourceDataset.drop(x_train.index)

# Split what's left in 2
# 100% - 80% = 20% / 2 = 10% of total dataset
validation_sample = splittingSourceDataset.sample(frac=0.5, random_state=0)
x_validation, y_validation = dataframeToXandY(validation_sample)

test_sample = splittingSourceDataset.drop(validation_sample.index)
x_test, y_test = dataframeToXandY(test_sample)
```

La fonction “dataframeToXandY” permet de séparer les labels des données. Les labels sont retournés sous la forme d’une matrice, grâce à l’encoder “LabelBinarizer” de “scikit-learn”.

```
def dataframeToXandY(dataframe):
   encoder = LabelBinarizer()
   y = dataframe['voyelle']
   transformedY = encoder.fit_transform(y)
   x = dataframe.drop(columns=['voyelle'])

   return (x, transformedY)
```

Gestion des paramètres des modèles

Afin de pouvoir tester plusieurs configurations de modèles, j'ai mis en place un système basé sur un fichier json.

Le fichier json contient les configurations sous la forme suivante :

```
{
   "models": [
       {
           "activation_function": "relu",
           "input_activation_function": "relu",
           "output_activation_function": "sigmoid",
           "loss": "binary_crossentropy",
           "optimizer": "adam"
       },
      ...
   ]
}
```

Une classe "ModelConfigReader" permet de charger ce fichier et en extraire les configurations. Il suffit ensuite d'itérer sur les configurations et de les appliquer au modèle keras.

Exemple :

```
for modelConfig in modelConfigReader.configs:
   ACTIVATION = modelConfig.activation_function
   INPUT_ACTIVATION = modelConfig.input_activation_function
   OUTPUT_ACTIVATION = modelConfig.output_activation_function

   model = Sequential()
   model.add(Dense(NUM_CLASSES, activation=INPUT_ACTIVATION,
input_dim=INPUT_DIMENSION))
   model.add(Dense(1024, activation=ACTIVATION))
   ...

   model.compile(
       loss=modelConfig.loss,
       optimizer=modelConfig.optimizer,
       metrics=["accuracy"])
```

Question 4

Réseau de neurones “feedforward”

Les premières versions du réseau de neurones que j’ai réalisé ont été de manière semblables à celle vue dans le TP1.

La structure du réseau est :

```
model = Sequential()
model.add(Dense(NUM_CLASSES, activation=INPUT_ACTIVATION, input_dim=INPUT_DIMENSION))
model.add(Dropout(0.3))
model.add(Dense(1024, activation=ACTIVATION))
model.add(Dropout(0.5))
model.add(Dense(512, activation=ACTIVATION))
model.add(Dropout(0.5))
model.add(Dense(256, activation=ACTIVATION))
model.add(Dropout(0.5))
model.add(Dense(NUM_CLASSES, activation=OUTPUT_ACTIVATION))
```

Les valeurs des constantes sont :

```
NUM_CLASSES = 10
INPUT_ACTIVATION, ACTIVATION = 'relu'
OUTPUT_ACTIVATION = 'sigmoid'
```

La fonction loss choisie est ‘binary_crossentropy’ et l’optimizer est ‘adam’.

Le résultat est :

```
--------------------RESULTS---------------------
activation        : relu
input activation  : relu
output activation : sigmoid
loss              : binary_crossentropy
optimizer         : adam

Test loss: 0.2941154440599146
Test accuracy: 0.8999963998794556
------------------------------------------------
```

Réseau de neurones récurrents

Le résultat ci-dessus ne me satisfaisant pas, j'ai décidé d'essayer d'utiliser un réseau de neurones récurrents. Ayant déjà vu ce type de réseaux dans différents articles, il est aussi utilisé dans un papier de recherche que j'ai lu durant la veille que j'ai effectuée sur le sujet. Dans leur papier "Formant Estimation and Tracking using Deep Learning", Yehoshua Dissen et Joseph Keshet décrivent avoir utilisé un réseau de neurones récurrents, composés de couches de neurones "Long Short Term Memory".

J'ai donc utilisé une architecture semblable à la leur.

```
rnnModel = Sequential()
rnnModel.add(Embedding(MAX_EMBEDDING_VALUE, NUM_CLASSES))
rnnModel.add(Dropout(0.3))
rnnModel.add(LSTM(512, activation=ACTIVATION, return_sequences=True))
rnnModel.add(Dropout(0.3))
rnnModel.add(LSTM(256, activation=ACTIVATION, return_sequences=True))
rnnModel.add(Dropout(0.3))
rnnModel.add(LSTM(128, activation=ACTIVATION))
rnnModel.add(Dropout(0.3))
rnnModel.add(Dense(NUM_CLASSES, activation='softmax'))
```

La fonction d'activation utilisée ici est 'relu' et 'softmax' en sortie, contrairement à leur choix qui est 'sigmoid'.

La valeur de la constance MAX_EMBEDDING_VALUE est la valeur maximum que peut atteindre une valeur dans notre jeu de données. La ligne de code ci-dessous se charge de récupérer cette valeur maximum.

```
NUM_CLASSES = len(audioDataPreprocessor.completeDataset.voyelle.unique())
```

Le résultat avec cette architecture est :

```
Test loss: 0.1993678035500293
Test accuracy: 0.9152666330337524
```

Quoique supérieure, la différence avec un réseau "feedforward" n'est pas énorme. Durant tous mes tests je n'ai pas réussi à dépasser ~90% de précision.