Bordeaux INP – ENSEIRB-MATMECA

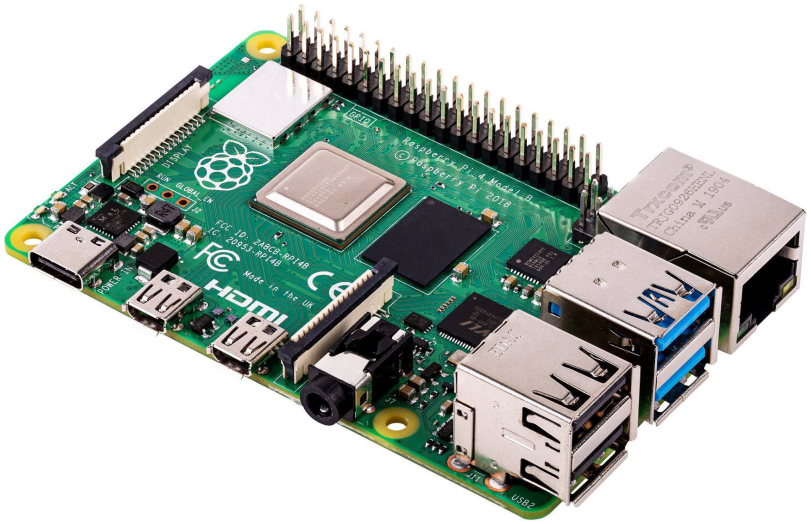
Filière Systèmes Électroniques Embarqués

ES0B UE SEE0-B - Modélisation Système (S10)

ES0EN343 Développement IA légère sur un système embarqué

Développement d'une IA embarquée sur Raspberry Pi

Adrien CLAIN et Clément SAVARY



Sommaire

Introduction

L'objectif de ce TP est de mettre en œuvre un algorithme d'intelligence artificielle légère sur un système embarqué. Les connaissances acquises devront permettre de développer une application embarquée de reconnaissance de caractères à partir d'une caméra.

Nous ferons face à différentes étapes pour l'implémentation de l'application embarquée:

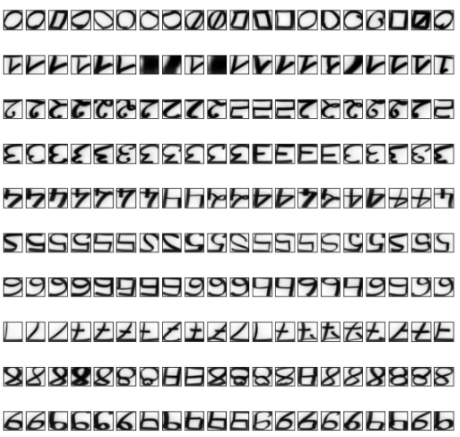
- Implémentation de la phase d'apprentissage avec le framework keras et le dataset fourni avec l'export des différentes couches du modèle
- Implémentation de la phase d'inférence sur carte
- Acquisition de nouvelles images et mesure de la performance de l'algorithme

I. Implémentation de la phase d'apprentissage

1. Création de la base de données

Afin de réaliser notre propre IA qui aura comme objectif la reconnaissance de chiffres manuscrits, nous allons utiliser le framework Keras. Celui-ci est une bibliothèque open source écrite en python, permettant d'interagir facilement avec des algorithmes de réseaux de neurones et d'apprentissage automatique, notamment TensorFlow.

La première étape de la phase d'apprentissage est de construire une base de données. Celle-ci est composée de 210 images de chiffres manuscrits de taille 28x28 pixel, 21 image par chiffre.



2. Importation des données

Nous commençons par sauvegarder nos données, qui seront utilisées pour la phase d'apprentissage.

Nous allons utiliser les 20 premières images de chaque chiffre comme source d'apprentissage, la 21ème image sera utilisée pour la phase d'inférence. A chaque image est associé un label qui correspond au chiffre représenté, celui-ci permet à l'IA de faire la correspondance entre l'image manuscrite et le résultat attendu. Attention les images doivent être convertie en niveau de gris, afin d'être utilisées par L'IA.

Importation des images d'apprentissage :

```
ImagePath=[]
train_labels=[]

for chiffre in range(0,10):#0 à 9
    for indice in range(0,20):#0 à 19

ImagePath.append('/content/drive/MyDrive/EN343/EN343_source/Serveur/Projet/
Database/Prof/Images/bmpProcessed/' + str (chiffre) +"_" +str
(indice)+".bmp")
        train_labels.append (chiffre)
#print(Labels)

train_labels_tab = keras.utils.to_categorical(train_labels,10)
print("\n train_labels :\n")
print(train_labels_tab)

train_images = [numpy.array(Image.open(str(l))) for l in ImagePath]
train_images = numpy.sum(train_images,axis=3)//3/255
```

Importation des images d'inférence :

```
test_images_Path=[]
test_labels=[]

for chiffre in range(0,10):

test_images_Path.append('/content/drive/MyDrive/EN343/EN343_source/Serveur/
Projet/Database/Prof/Images/bmpProcessed/' + str (chiffre) +"_" +str
(20)+".bmp")
        test_labels.append (chiffre)
#print(test_labels)
#print(test_images)
test_labels_tab = keras.utils.to_categorical(test_labels,10)
print("\n test_labels :\n")
print(test_labels_tab)

test_images = [numpy.array(Image.open(str(l))) for l in test_images_Path]
test_images = numpy.sum(test_images,axis=3)//3/255
```

Images avec le label associé :

3. Construction du modèle

Nous allons créer un modèle composé de 2 couches de neurones. La couche d'entrée est composée de 128 neurones avec une fonction activation de type "relu", celle-ci permet de supprimer les régressions et de conserver les ressemblances de pattern. Enfin la couche de sortie possède 10 neurones, chaque neurone continent un score de probabilité de détection d'un chiffre. Enfin, la fonction softmax permet d'avoir une probabilité normalisée entre 0 et 1.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10,),
    tf.keras.layers.Activation("softmax")
])
```

4. Former le Modèle

Une fois le modèle construit nous pouvons démarrer la phase d'apprentissage de la manière suivante :

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])

model.fit(train_images, train_labels_tab, epochs=8)
```

```
Epoch 1/8
/usr/local/lib/python3.7/dist-packages/tensorflow/python/util/dispatch.py:1082: UserWarning:
  return dispatch_target(*args, **kwargs)
7/7 [==============================] - 1s 4ms/step - loss: 2.2544 - accuracy: 0.2150
Epoch 2/8
7/7 [==============================] - 0s 4ms/step - loss: 1.3313 - accuracy: 0.7000
Epoch 3/8
7/7 [==============================] - 0s 3ms/step - loss: 0.8638 - accuracy: 0.8350
Epoch 4/8
7/7 [==============================] - 0s 3ms/step - loss: 0.5930 - accuracy: 0.8750
Epoch 5/8
7/7 [==============================] - 0s 3ms/step - loss: 0.4251 - accuracy: 0.9150
Epoch 6/8
7/7 [==============================] - 0s 3ms/step - loss: 0.3239 - accuracy: 0.9300
Epoch 7/8
7/7 [==============================] - 0s 3ms/step - loss: 0.2449 - accuracy: 0.9650
Epoch 8/8
7/7 [==============================] - 0s 3ms/step - loss: 0.1943 - accuracy: 0.9750
<keras.callbacks.History at 0x7f297886e710>
```

Nous utilisons un optimiseur de type “adam” pour la descente de gradient. Nous nourrissons le modèle avec les images d'entraînements, et effectuons 8 itérations. Nous obtenons une précision de 97% ce qui est suffisant pour passer à la phase d’inférence. Le but étant de ne pas avoir une précision de 100% car le modèle serait sur-entraîné.

5. Evaluation de la précision

Nous mesurons le performance du système avec les images de test :

```
test_loss, test_acc = model.evaluate(test_images,  test_labels_tab, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
1/1 - 0s - loss: 0.1791 - accuracy: 1.0000 - 19ms/epoch - 19ms/step

Test accuracy: 1.0
```

Toutes les images de test ont été correctement interprétées.

6. Prédictions

```
probability_model = tf.keras.Sequential([model,
                                         tf.keras.layers.Softmax()])

predictions = probability_model.predict(test_images)

predictions[0]
```

```
array([0.16318303, 0.08870874, 0.08882634, 0.08889688, 0.08879302,
       0.08890814, 0.11704966, 0.08869775, 0.09196321, 0.09497321],
      dtype=float32)
```

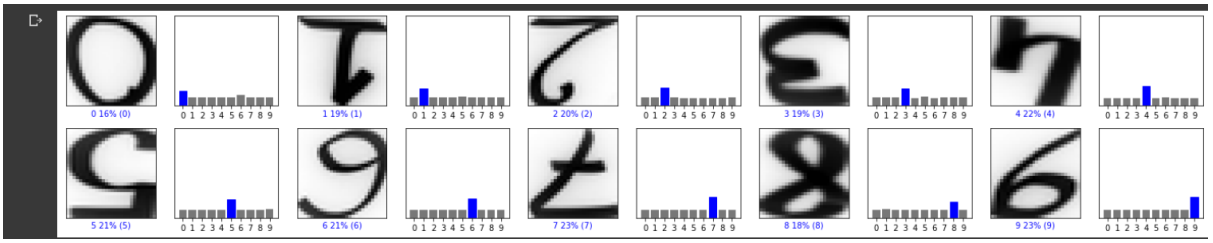
Nous affichons la probabilité de détection d'un chiffre, ici le 0. Nous pouvons aussi directement afficher le résultat en trouvant la valeur max du tableau.

```
numpy.argmax(predictions[0])
```

```
0
```

Voici les différents chiffres avec leur prédictions :

```
# Plot the first X test images, their predicted labels, and the true
labels.
# Color correct predictions in blue and incorrect predictions in red.
num_rows = 2
num_cols = 5
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_image(i, predictions[i], test_labels, test_images)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```



7. Enregistrement des poids et et biais

Une fois L'IA correctement entraînée, nous enregistrons les poids et les biais des couches 1 et 2 dans des fichiers texte, afin de pouvoir implémenter celle-ci en code C, afin d'être utilisée sur système embarqué comme une raspberry.

```
weight1, biais1 =model.layers[1].get_weights()
weight2, biais2 =model.layers[2].get_weights()

weight1 = numpy.array(weight1)
weight2 = numpy.array(weight2)
numpy.savetxt("./drive/MyDrive/EN343/poids_1.txt",weight1.T)
numpy.savetxt("./drive/MyDrive/EN343/poids_2.txt",weight2.T)
numpy.savetxt("./drive/MyDrive/EN343/biais_1.txt",numpy.array(biais1))
numpy.savetxt("./drive/MyDrive/EN343/biais_2.txt",numpy.array(biais2))
```

II. Implémentation de la phase d'inférence

Nous avons extrait les poids et biais du modèle dans des fichiers textes et maintenant nous allons exploiter ces fichiers dans le but d'implémenter la phase d'inférence avec le langage C.

1. Acquisition des poids et des biais

Nous commençons par ouvrir les fichiers en question :

```
OpenFile(&ppoid1,"files/poids_1.txt");
OpenFile(&ppoid2,"files/poids_2.txt");
OpenFile(&pbiais1,"files/biais_1.txt");
OpenFile(&pbiais2,"files/biais_2.txt");
```

Pour rappel, la première couche contient 128 neurones et la taille de l'image traitée est de 28x28. Nous aurons donc 784 valeurs par neurones en sortie. Les poids seront donc stockés dans un tableau 2D de taille 128x784 :

```
  for(int i=0; i<128;i++)
    {
      for(int j=0; j<784;j++)
      {
          fscanf(ppoid1,"%f ",&carac);
          poid1[i][j]=carac;
      }
    }
fclose(ppoid1);
```

Même chose pour la deuxième couche sauf que nous avons maintenant 10 neurones qui recevront les résultats de la couche précédente soit 128 valeurs :

```
 for(int i=0; i<10;i++)
   {
     for(int j=0; j<128;j++)
     {
        fscanf(ppoid2,"%f ",&carac);
        poid2[i][j]=carac;
     }
   }
   fclose(ppoid2);
```

Nous stockons maintenant les biais aux couches respectives dans des tableaux de tailles 128 et 10 correspond aux nombre de neurones de chaque couche:

```
for(int i = 0 ;i<128;i++){
   fscanf(pbiais1,"%f ",&carac);
   biais1[i]=carac;
 }
 fclose(pbiais1);

 for(int i = 0 ;i<10;i++){
   fscanf(pbiais2,"%f ",&carac);
   biais2[i]=carac;
 }
 fclose(pbiais2);
```

Maintenant que nous avons stocké toutes les données nécessaires à notre algorithme nous pouvons passer à la phase de calcul et de prédiction des valeurs.

2. Phase de calcul et prédiction

Pour cette phase, nous déclarons trois fonctions dans le fichier neurone.h :

```
#ifndef NEURONE_H
#define NEURONE_H

void calcul_neurone(float poids[][784],int nbpoid, int nb_neural, float *biais, float
*pixel, float *y_neural);
void relu(float *donnee, int nb_donnee);
void softmax(float *input, int input_len);

#endif
```

La fonction `calcul_neurone()` va permettre de réaliser l'algorithme suivant :

$$y_{neural} = f\left(\left(\sum_{i\in\mathbb{N}} w_i * x_i\right) + b\right)$$

Avec :

- y_neural : résultat de l'équation
- w : poids de la couche
- x : pixel de l'image
- b : biais de la couche

Nous vectorisons les pixels de l'image pour faciliter le traitement avec la fonction `LoadImageInVector()` :

```
void LoadImageInVector(float *tab_image_vecteur, int len,char *nom_fichier)
{
  BMP bitmap;
  FILE *pImage=NULL;
  OpenFile(&pImage,nom_fichier);
  LireBitmap(pImage, &bitmap);
  fclose(pImage);
  ConvertRGB2Gray(&bitmap);
   for(int i = 0 ;i<28;i++){
      for(int j = 0 ;j<28;j++){
        tab_image_vecteur[i*28+j]= (float)(bitmap.mPixelsGray[i][j])/255;

      }
   }
    DesallouerBMP(&bitmap);
}
```

Nous vectorisons ainsi l'image rentrée en argument lors de l'éxecution du programme :

```
   char *nom_fichier;
   nom_fichier=argv[1];
   LoadImageInVector(tab_image_vecteur_float,784,nom_fichier);
```

Avec ceci nous avons tout le nécessaire pour réaliser notre algorithme. Pour cela, nous avons mis en place la fonction `calcul_neurone` (neurone.c) qui va prendre en argument le poids, le biais et le pixel associé. La fonction va remplir le tableau `layer1_neural` de tous ces résultats :

```
void calcul_neurone(float poids[][784],int nbpoid, int nb_neural, float
*biais, float *pixel, float *y_neural)
{
    for (int y = 0 ; y < nb_neural ; y++)
    {
        y_neural[y] =0;
        for (int i = 0 ; i < nbpoid ; i++)
        {
            y_neural[y] += poids[y][i]*pixel[i];
        }
    y_neural[y] += biais[y];
    }
}
```

Ensuite la fonction relu va permettre rejeter tous les résultats avec une valeur négative afin de rejeter toute régression (différence) et donc de ne garder que les pattern positif (ressemblance).

```
void relu(float *donnee, int nb_donnee)
{
    for (int i = 0 ; i < nb_donnee - 1 ; i++)
    {
        if ( donnee[i] < 0 )
        {
            donnee[i] = 0;
        }
    }
}
```

Ensuite après la deuxième couche de neurone, nous devons utiliser la fonction softmax dans le but de convertir un score en probabilité dans un contexte de classification multi-classe. Celle-ci suit la loi suivante :

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ pour tout } j \in \{1, \dots, K\}.$$

Nous pouvons décrire cette équation avec la fonction suivante :

```
void softmax(float* input, int size) {

    int i;
    float m, sum, constant;

    m = -INFINITY;
    for (i = 0; i < size; ++i) {
        if (m < input[i]) {
            m = input[i];
        }
    }

    sum = 0.0;
    for (i = 0; i < size; ++i) {
        sum += exp(input[i] - m);
    }

    constant = m + log(sum);
    for (i = 0; i < size; ++i) {
        input[i] = exp(input[i] - constant);
    }

}
```

Maintenant que nous avons déclaré et définit nos fonction de calcul, nous pouvons les utiliser dans cet ordre ci :

- Calcul couche neurone n°1
- Fonction relu
- Calcul couche neurone n°2
- Fonction softmax

```
calcul_neurone(poid1,784,128,biais1,tab_image_vecteur_float,layer1_neural);
relu(layer1_neural,128);
calcul_neurone(poid2,128,10,biais2,layer1_neural,layer2_neural);
softmax(layer2_neural,10);
```

Maintenant nous avons les résultats stockés dans le tableau `layer2_neural` qui contient les probabilité de correspondance de chaque chiffre. Il reste maintenant plus qu'à idnentifer la plus grande probabilité de ressemblance et à l'afficher sur le terminal.

```
float probaMax =0;
 int valeurChiffre=0;
 for(int i = 0 ;i<10;i++){
   if(layer2_neural[i]> probaMax){
      probaMax =layer2_neural[i];
      valeurChiffre =i;
   }
 }
 printf("Valeur Prédite est : %d avec une proba de %f\n ",valeurChiffre,probaMax);
```

En prenant l'exemple de l'exécution de la reconnaissance d'une image du chiffre 0. Nous lançons l'exécution avec la commande ci-dessous :

```
$ ./all ../../Database/Prof/Images/bmp_ProcessedSeuil/0_2.bmp
```

Nous obtenons la bonne prédiction et avec sa probabilité :

```
Valeur Prédite est : 0 avec une proba de 0.925138
```

Maintenant nous allons réaliser les prédictions sur une Raspberry Pi .

III. Acquisition de nouvelle images et mesures de performances

Après avoir réalisé tous les codes C sur notre PC, nous allons l'intégrer sur la carte Raspberry avec le protocole SSH.
Pour cela, nous copions notre dossier de travail :

```
$ scp -r -P 2223 Project pi_12@176.179.183.226:/.
```

Nous nous assurons que le programme actuel s'exécute correctement.

```
Valeur Prédite est : 0 avec une proba de 0.925138
pi_12@raspberry:~/Projet/codeC/Inference $
```

Nous testons ensuite la fonctionnalité de prise de photo avec la caméra intégrée à la carte. Notre image devra être de 64x64 :

```
$ raspistill -e bmp -w 64 -h 64 -o photo1.bmp
```

Nous vérifions l'état de l'image en la copiant sur notre disque local et en l'ouvrant :

```
$  scp -P 2223 pi_12@176.179.183.226:~/…../photo1.bmp .
```

Nous remarquons que l'image est de taille 64x64 alors que notre code fonctionne avec une image de 28x28 et elle contient plusieurs chiffres ce qui va gêner la phase de prédiction. Pour contrer cela, un fichier traitement.c nous est fourni pour pouvoir traiter cette image en la rognant.
Néanmoins, le code nous donne l'image ci-dessous :

Conclusion

Dans ce TP, nous avons vu le fonctionnement d'une intelligence artificielle du point théorique mais aussi pratique. La phase d'apprentissage réalisée en python nous a permis d'extraire les paramètres importants pour la mise en place des couches de neurones, c'est-à-dire l'export des poids et biais dans des fichiers texte.

Ces poids et biais vont être utiles pour la phase d'inférence réalisée en langage C pour le côté embarqué. En effet, nous avons à notre disposition une carte Raspberry Pi connectée à un réseau afin que nous puissions s'y connecter par le biais du protocole sécurisé SSH. Nous y avons intégré notre code embarqué et testé la prédiction de chiffre sur la carte avec des images mises à disposition.

Nous avons réussi à prendre des photos avec la caméra intégrée à la carte Raspberry mais le traitement de l'image n'est pas concluant pour que la prédiction puisse être optimale.