Monest-ai : Documentation Technique

TensorFlowModel

Le TensorFlowModel est une classe python permettant de construire, compiler, entrainer, prédire et sauvegarder des modèles d'apprentissage automatique basés sur la librairie Keras de TensorFlow.

Cette dernière hérite de la classe Model qui a le rôle d'interface ou l'on retrouve les prototypes des méthodes ainsi que les attributs d'instance de la classe TensorFlowModel.

Attributs de Classe :

- __activations : Liste des fonctions d'activation (relu, softmax, tanh, sigmoid, linear)

```
# Class parameters and constants
__activations = ["relu", "softmax", "tanh", "sigmoid", "linear"]
```

Attributs d'Instance :

- _has_file (bool) : Indique si le modèle possède un fichier.
- _model (keras.Model) : Objet modèle Keras sous-jacent chargé à partir d'un fichier.
- _save (bool) : Indique si le modèle doit être sauvegardé.
- _file_name (str) : Nom de fichier pour la sauvegarde du modèle.
- _path (str) : Chemin d'accès pour la sauvegarde du modèle.
- _compiled (bool) : Indique si le modèle est compilé.
- _layers (list) : Liste des couches du modèle à construire (Dense ou Flatten).

Méthodes :

- __init__(self, file=None, save=False, name=None, path=None) :
  - Initialise l'instance de la classe.
  - Charge un modèle existant à partir d'un fichier si ce dernier est fourni.

- Initialise un modèle vide Keras Sequential si aucun fichier n'est fourni.
- Définit les attributs _save, _file_name, _path et _compiled en fonction des paramètres.

```
def __init__(self, file = None, save = False, name = None, path = None):
    if file != None:
        self._has_file = True
        self._model = keras.models.load_model(file)
    else:
        self._model = keras.models.Sequential()
    if save:
        self._save = save
        self._file_name = name
        self._path = path
        self._compiled = True
```

- add_layer(self, layer_type, datas) :
  - Ajoute une couche au modèle en construction.
  - Vérifie si un modèle n'est pas déjà chargé.
  - Valide le type de couche (TensorflowEnum.DENSE ou TensorflowEnum.FLATTEN).
  - Pour les couches DENSE :
    - Valide le nombre de neurones et l'activation.
    - Lève une exception InvalidNeurons si le nombre de neurones est supérieur aux couches précédentes.
    - Lève une exception WrongActivation si l'activation n'est pas dans la liste __activations et donc inconnue.
  - Pour les couches FLATTEN :
    - Valide la forme de l'entrée.

- Lève une exception InvalidType si le type de couche n'est pas reconnu.

```python
def add_layer(self, layer_type, datas):
    if not self._has_file:
        if layer_type == TensorflowEnum.DENSE:
            neurons = datas["neurons"]
            activation = datas["activation"]
            if activation in self.__activations:
                if len(self._layers) > 0:
                    for layer in self._layers:
                        if layer["type"] == "Dense":
                            if layer["neurons"] <= neurons:
                                raise InvalidNeurons("Too many neurons for this layer", neurons)
                self._layers.append({"neurons": neurons, "activation": activation, "type": "Dense"})
            else:
                WrongActivation("Undefined activation", activation)
        elif layer_type == TensorflowEnum.FLATTEN:
            shape = datas["shape"]
            self._layers.append({"shape": shape, "type": "Flatten"})
        else:
            raise InvalidType("Layer type is not defined")
    else:
        raise LoadedModel("A model is loaded")
```

- compile(self, loss="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"]) :
  - Compile le modèle pour l'apprentissage.
  - Vérifie si un modèle n'est pas déjà chargé.
  - Vérifie s'il y a au moins une couche définie.
  - Lève une exception NoLayers s'il n'y a pas de couches.
  - Construit le modèle Keras en ajoutant les couches définies avec add_layer.
  - Compile le modèle Keras.
  - Définit l'attribut _compiled à True.

```python
def compile(self, loss="sparse_categorical_crossentropy", optimizer="sgd", metrics=["accuracy"]):
    if not self._has_file:
        if len(self._layers) > 0:
            for layer in self._layers:
                if layer["type"] == "Dense":
                    self._model.add(tf.keras.layers.Dense(layer["neurons"], activation=layer["activation"]))
                elif layer["type"] == "Flatten":
                    self._model.add(tf.keras.layers.Flatten(input_shape=layer["shape"]))
            self._model.compile(loss=loss, optimizer=optimizer, metrics=metrics)
            self._compiled = True
        else:
            raise NoLayers("No layers found")
    else:
        raise LoadedModel("A model is loaded")
```

- fit(self, values, labels, epochs) :
  - Entraîne le modèle sur les données fournies.
  - Vérifie si le modèle est compilé ou chargé.
  - Lève une exception NoCompilation si le modèle n'est pas compilé.
  - Entraîne le modèle Keras avec les données d'entrée Stocke l'historique d'entraînement dans l'attribut _history.

```
def fit(self, values, labels, epochs):
    if self._compiled or self._has_file:
        self._history = self._model.fit(values, labels, epochs=epochs)
    else:
        raise NoCompilation("Model is not compiled")
```

- predict(self, value) :
    - Effectue une prédiction sur une nouvelle donnée.
    - Vérifie si le modèle est compilé ou chargé.
    - Lève une exception NoCompilation si le modèle n'est pas compilé.
    - Utilise le modèle Keras pour prédire la sortie sur la donnée.
    - Renvoie la prédiction.

```
def predict(self, value):
    if self._compiled or self._has_file:
        return self._model.predict(value)
    else:
        raise NoCompilation("Model is not compiled")
```

- save(self) :
    - Sauvegarde le modèle.
    - Vérifie si la sauvegarde est activée (_save) et si le modèle est compilé ou chargé.
    - Lève une exception NoSaveMode si la sauvegarde n'est pas activée.
    - Sauvegarde le modèle Keras dans le chemin _path avec le nom _file_name.

```
def save(self):
    if self._save and (self._compiled or self._has_file):
        self._model.save(self._path + "/" + self._file_name + ".h5")
    else:
        raise NoSaveMode("This model is started in no save mode")
```

Request

La classe Request implémente toute les méthodes HTTP (GET, POST, PUT, PATCH, DELETE) permettant de mettre en place différentes requêtes.

Voici un exemple d'implémentation d'un requête POST :

```python
def post_request(self, url, params=None, headers=None, auth=None, data=None):
    response = requests.post(url, params=params, headers=headers, auth=auth, data=data)
    if response.ok:
        value = json.loads(response.text)
        return value
    else:
        return {}
```

DatasetJSON

Cette classe a pour but de charger et de traiter un ensemble de données au format JSON pour l'apprentissage d'un modèle.

Cette dernière hérite de la classe DataSet qui a le rôle d'interface, de la manière que pour la classe TensorFlowModel cette dernière contient les prototypes des méthodes ainsi que les attributs d'instance de la classe DataSetJSON.

Attributs d'Instance :

- _file (str) : Chemin d'accès au fichier JSON contenant les données.
- _datas (list) : Liste des dictionnaires chargés à partir du fichier JSON.

Méthodes :

- __init__(self, file) :
    - Initialise l'instance de la classe.
    - Stocke le chemin d'accès au fichier JSON dans _file.
    - Appelle la méthode privée __get_file_content pour charger le contenu du fichier.

```python
def __init__(self, file):
    self._file = file
    self.__get_file_content()
```

- __get_file_content(self) :
    - Méthode privée pour charger le contenu du fichier JSON.
    - Vérifie si le fichier existe.
    - Lève une exception UnreadFile si le fichier n'est pas trouvé.
    - Ouvre le fichier en lecture (mode "r") avec encodage UTF-8.
    - Lit le contenu du fichier.

- Ferme le fichier.
- Charge le contenu JSON dans l'attribut _datas.

```
def __get_file_content(self):
    if os.path.isfile(self._file):
        opened_file = open(self._file, "r", encoding="utf-8")
        content = opened_file.read()
        opened_file.close()
        self._datas = json.loads(content)
    else:
        raise UnreadFile("File not found", self._file)
```

- get_datas(self) :
  - Renvoie la liste des dictionnaires chargés à partir du fichier JSON.

```
def get_datas(self):
    return self._datas
```

- get_shape_values(self, n) :
  - Prépare les données pour l'entraînement en les transformant en un tableau NumPy.
  - n représente le nombre de valeurs par échantillon.
  - Calcule le nombre de blocs de données à créer en fonction de la longueur de _datas et de n.
  - Crée un tableau NumPy vide de forme (nb, n) et de type float32.
  - Parcourt chaque bloc de données :
    - Parcourt chaque valeur dans le bloc :
      - Si l'index est inférieur à la longueur de _datas :
        - On récupère la valeur "data_esp32" du dictionnaire courant et on l'insère dans le tableau.
        - On incrémente l'index.
  - Normalise les données en soustrayant la moyenne et en divisant par l'écart type.
  - Renvoie le tableau NumPy normalisé.

```
def get_shape_values(self, n):
    nb = len(self._datas) // n

    if len(self._datas) % n != 0:
        nb += 1
    metadata = np.ndarray(shape=(nb, n), dtype=np.float32)
    metadata.fill(0)
    index = 0
    for i in range(0, len(metadata)):
        for x in range(0, len(metadata[i])):
            if index < len(self._datas):
                metadata[i].put(x, self._datas[index]["data_esp32"])
                index += 1
    return (metadata - np.mean(metadata)) / np.std(metadata)
```

- get_label_values(self) :
    - Récupère les labels des données.
    - Crée un tableau NumPy vide de forme (len(self._datas)) et de type int.
    - Parcourt chaque dictionnaire dans _datas :
        - Récupère le label en utilisant la clé "name" du dictionnaire et la constante __labels (contenant les différents “name” que peut avoir un dictionnaire).
        - Insère le label dans le tableau NumPy.
    - Renvoie le tableau NumPy contenant les labels.

```
def get_label_values(self):
    data = np.ndarray(shape=(len(self._datas)), dtype=int)
    data.fill(0)

    for i in range(0, len(self._datas)):
        dic = self._datas[i]
        data.put(i, self.__labels.get(dic["name"]))
    return data
```

- timestamp_values(self) :
    - Récupère les timestamps (horodatages) des données.
    - Parcourt chaque dictionnaire dans _datas :
        - Récupère la valeur "date_esp32" du dictionnaire et l'ajoute à la liste.
    - Renvoie la liste des timestamps.

```python
def timestamp_values(self):
    data = []

    for dic in self._datas:
        data.append(dic['date_esp32'])
    return data
```

Main

La gestion du point d'entrée est assez simple on crée le model soit à partir de fichiers soit en instanciant notre réseau de neurones.

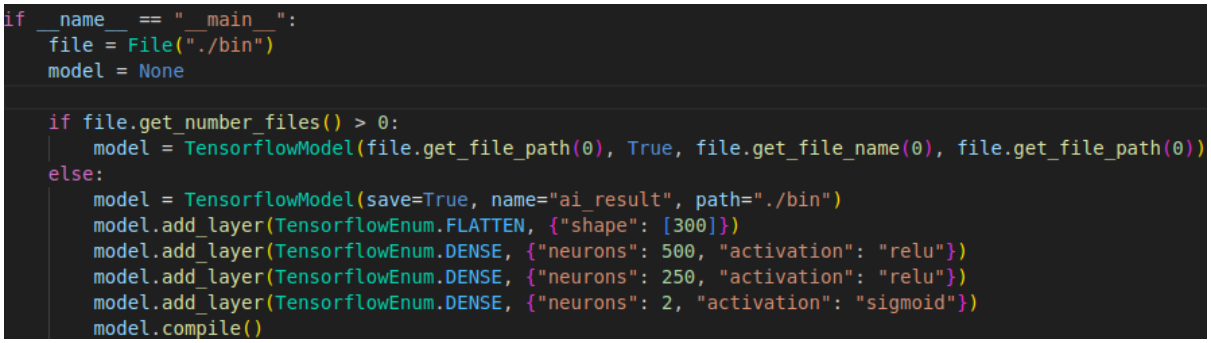
Initialisation des Variables :

- File : Objet de la classe File initialisé avec le chemin "./bin".
- Model : Variable initialisée à None pour stocker l'objet TensorflowModel.

Vérification de la présence de fichiers dans le dossier "./bin" :

- La méthode get_number_files de l'objet file est appelée pour vérifier s'il y a des fichiers dans le dossier "./bin".
- Si le nombre de fichiers est supérieur à 0 :
    - Un modèle TensorflowModel est chargé à partir du premier fichier trouvé dans "./bin" en utilisant les méthodes get_file_path et get_file_name de l'objet file.
    - Le paramètre save du constructeur TensorflowModel est défini à True pour indiquer que le modèle doit être sauvegardé après modification.
    - Les autres paramètres du constructeur définissent le nom du fichier et le chemin de sauvegarde en cas de modification du modèle.
- Sinon (s'il n'y a pas de fichier) :
    - Un nouveau modèle TensorflowModel est créé sans fichier de sauvegarde initial.
    - Le paramètre save du constructeur est défini à True pour activer la sauvegarde.
    - Le nom du fichier de sauvegarde est défini à "ai_result".
    - Le chemin de sauvegarde est défini à "./bin".

- La méthode add_layer du modèle est appelée pour définir l'architecture du réseau de neurones :
    - Une couche FLATTEN est ajoutée en premier pour aplatir les données d'entrée.
    - Deux couches DENSE cachées sont ajoutées avec respectivement 500 et 250 neurones et l'activation "relu".
    - Une couche de sortie DENSE est ajoutée avec 2 neurones et l'activation "sigmoid" pour la classification binaire.
- Enfin, la méthode compile du modèle est appelée pour compiler le modèle d'apprentissage automatique prêt à être entraîné.



```
if __name__ == "__main__":
    file = File("./bin")
    model = None

    if file.get_number_files() > 0:
        model = TensorflowModel(file.get_file_path(0), True, file.get_file_name(0), file.get_file_path(0))
    else:
        model = TensorflowModel(save=True, name="ai_result", path="./bin")
        model.add_layer(TensorflowEnum.FLATTEN, {"shape": [300]})
        model.add_layer(TensorflowEnum.DENSE, {"neurons": 500, "activation": "relu"})
        model.add_layer(TensorflowEnum.DENSE, {"neurons": 250, "activation": "relu"})
        model.add_layer(TensorflowEnum.DENSE, {"neurons": 2, "activation": "sigmoid"})
        model.compile()
```