JIGSAW MULTILINGUAL TOXIC COMMENT CLASSIFICATION

PAR :

NAMA NYAM GUY ANTHONY

24 Juin 2020

Résumé de travail

Le projet de la compétition ***Kaggle*** a porté sur la conversation en IA. Plus spécifiquement, il s'agissait de prédire la probabilité qu'un commentaire soit toxique. La donnée d'entraînement contient une colonne de commentaires ***comment_text*** fournies en anglais provenant du *Wikipedia talk page comments* et *Civil Comments*. Les commentaires de la donnée d'entraînement ont été classés comme toxique ou pas(0/1) dans la colonne ***toxic***. La donnée de test contient une colonne de commentaires ***comment_text*** composés de plusieurs différentes langues.

Dans ce contexte multilingue, nous avons utilisé les deux modèles qui supportent le mieux actuellement de multiple langages à savoir ***Bert*** et ***XLM***. Le modèle ***bert_multi_cased_L-12_H-768_A-12/2*** au format *TF2 SavedModel* de *TensorFlowhub* sera utilisé comme modèle *Bert*. En ce qui concerne *XLM*, nous avons utilisé le modèle ***XLM-Roberta*** de *HuggingFace*. Il faut dire que nous avons utilisé ces modèles entraînés sur les larges dataset.

L'étape suivante d'*encodage* consiste à fournir une représentation vectorielle dense pour le langage naturel. Cette opération se déroule en deux principales tâches dont chaque modèle offre des api à leur réalisation :

1. *Tokenisation* : prend en entrée du texte et renvoie les tokens constituant l'entrée
2. *Encodage* : prend en entrée les tokens de la tâche de Tokenisation et renvoie la représentation vectorielle de chaque token dans son contexte.

Le méthode de tokenisation du modèle de bert est *tokenize* et la méthode d'encodage *convert_tokens_to_ids*. Dans le modèle XLM-Roberta, ces deux tâches sont regroupées dans la méthode *encode_plus* de l'objet tokenizer. Cette étape est appliquée à l'ensemble du jeu de données.

A ce niveau, il s'agit d'adapter ou construire le modèle sous jacent à notre problème(*transfer learning*). L'architecture *transformer* utilisée à laquelle nous ajustons la partie classifier ainsi la forme des entrées du modèles. Avant de passer à l'apprentissage, on compile le modèle en lui fournissant la fonction de perte , l'optimiseur, ainsi la métrique à évaluer. Tout ce qui précède se passe sous la portée ***TPU***(Utilisation du TPU).
On entraîne chaque modèle dans un premier temps sur le jeu de donné d'entraînement, puis sur le jeu de donnée de validation qui est multilingue. Ceci se faisant en précisant le nombre d'étape par époque ainsi le nombre d'époque.

M-BERT

Nous avons la fonction ***get_tokenizer*** qui renvoie le tokenizer *Bert* multilingue. Cette fonction prend en entrée le modèle *Bert* au format *TF2 SavedModel* qui exige TensorFlow 2 and TensorFlow Hub. L'instanciation de l'objet de la classe *FullTokenizer* attend le vocabulaire(vocab_file) stocké dans *tf.saved_model.Asset* et le flag(do_lower_case) stocké dans *tf.variable* de l'objet *SavedModel*.

Paramètres : bert_path : str
Chemin vers Bert Saved Model

Retour : tokenizer : Objet bert.tokenization.FullTokenizer
Tokenizer bert

```
def get_tokenizer(bert_path=BERT_PATH_SAVEDMODEL):
    """Obtenez le tokenizer pour une couche BERT."""
    bert_layer = tf.saved_model.load(bert_path)
    bert_layer = hub.KerasLayer(bert_layer, trainable=False)
    vocab_file = bert_layer.resolved_object.vocab_file.asset_path.numpy()
    cased = bert_layer.resolved_object.do_lower_case.numpy()
    tf.gfile = tf.io.gfile  # bert.tokenization.load_vocab dans tokenizer
    tokenizer = bert.tokenization.FullTokenizer(vocab_file, cased)

    return tokenizer

tokenizer = get_tokenizer()
```

La fonction ***process_sentence*** renvoie la représentation vectorielle(input_ids) de chaque token dans son contexte. Cette fonction prend en entrée le texte à encoder, la taille maximale de la séquence à encoder et enfin le *tokenizer*. La fonction commence par tokeniser le texte, extrait la séquence maximale(moins 2) depuis le début à laquelle on n'a rajouté des tokens spéciaux puis on appelle la fonction *convert_tokens_to_ids*.

Paramètres : sentence : str
Texte à encoder

max_seq_length : int
Longueur maximale de la séquence encodée

tokenizer : Objet bert.tokenization.FullTokenizer
Tokenizer bert

Retour : input_ids : list
Liste contenant ids des tokens

input_mask : list
Liste contenant 1 pour les tokens et 0 pour le rembourrage

segment_ids : list
Liste de zéros

```
def process_sentence(sentence, max_seq_length=SEQUENCE_LENGTH, tokenizer=tokenizer):
    """Convertit la phrase sous la forme ['input_word_ids', 'input_mask', 'segment_ids']."""
    # Tokenize, et tronque à max_seq_length si necessaire.
    tokens = tokenizer.tokenize(str(sentence))
    if len(tokens) > max_seq_length - 2:
        tokens = tokens[:(max_seq_length - 2)]

    # Convertir les tokens de la phrase en IDs
    input_ids = tokenizer.convert_tokens_to_ids(["[CLS]"] + tokens + ["[SEP]"])

    # 1 pour les vrais tokens et un 0 pour les tokens de rembourrage.
    input_mask = [1] * len(input_ids)

    # Compléter par des zéros si la séquence est inférieur à max_seq_length
    pad_length = max_seq_length - len(input_ids)
    input_ids.extend([0] * pad_length)
    input_mask.extend([0] * pad_length)

    # Nous avons un seul segment d'entrée
    segment_ids = [0] * max_seq_length

    return (input_ids, input_mask, segment_ids)
```

La fonction ***multilingual_bert_model*** construit et renvoie le modèle *bert*(architecture). Le modèle constitué de trois entrées, des couches de l'architecture *transformer* de *bert* auquel on rajoute la partie classifieur.

Paramètres : max_seq_len : int

Utiliser pour la forme des entrées du modèle.

trainable_bert : bool

Entraîner la partie représentation ou pas.

Retour : _ : tf.keras.Model

Modèle *bert*

```
def multilingual_bert_model(max_seq_length=SEQUENCE_LENGTH, trainable_bert=True):
    """Construit et retourne un modèle Bert multilingue"""
    input_word_ids = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="input_word_ids")
    input_mask = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="input_mask")
    segment_ids = tf.keras.layers.Input(shape=(max_seq_length,), dtype=tf.int32, name="all_segment_id")

    bert_layer = tf.saved_model.load(BERT_GCS_PATH_SAVEDMODEL)
    bert_layer = hub.KerasLayer(bert_layer, trainable=trainable_bert)

    pooled_output, _ = bert_layer([input_word_ids, input_mask, segment_ids])
    output = tf.keras.layers.Dense(32, activation='relu')(pooled_output)
    output = tf.keras.layers.Dense(1, activation='sigmoid', name='labels')(output)

    return tf.keras.Model(inputs={'input_word_ids': input_word_ids,'input_mask': input_mask,
                                  'all_segment_id': segment_ids},outputs=output)
```

Dans cette partie, nous avons chargé et compilé le modèle *bert* sur ***TPU***. La fonction ***compile*** prend en paramètre la perte ***BinarCrossentropy*** lié au problème donc le label est binaire. L'optimiseur ***Adam*** initialisé avec pour *learning rate* à $1e-4$. La métrique AUC(aire au dessous de la courbe) à évaluer.

```
with strategy.scope():
    multilingual_bert = multilingual_bert_model()
    # Compile le modèle.
    multilingual_bert.compile(loss=tf.keras.losses.BinaryCrossentropy(),
                              optimizer=tf.keras.optimizers.Adam(lr=1e-4),
                              metrics=[tf.keras.metrics.AUC()])
```

La dernière étape consiste à l'apprentissage du modèle puis de prédictions sur le jeu de test.
Nous avons d'abord entraîné notre modèle sur le jeu d'entraînement qui est en anglais puis sur le jeu de validation qui est dans différentes langues. La fonction ***fit*** prend principalement les trois paramètres suivants : le jeu d'entraînement(*english_train_dataset, nonenglish_train_dataset*), le nombre d'étape par époque *steps_per_epoch*, et le nombre d'époque *epochs*

La fonction ***predict*** qui prend en entrée le jeu de test à prédire et renvoie les probabilités de chaque instance dans ce dataset.

```
# Train on dataset.
multilingual_bert.fit(english_train_dataset, steps_per_epoch=4000, epochs=2, verbose=1,
                      validation_data=nonenglish_val_datasets['Combined'], validation_steps=100)

multilingual_bert.fit(nonenglish_val_datasets['Combined'], steps_per_epoch=100, epochs=2)

# Prédictions
probabilities = np.squeeze(multilingual_bert.predict(test_sentences_dataset))
```

XLM-ROBERTA

Nous avons commencé par la configuration un ensemble de variables telles que *AUTO* qui prépare automatiquement les données du dataset à l'avance, la taille du batch, le nombre d'époque, la longueur d'encodage des séquences, le modèle à utiliser.

```
# Configuration

AUTO = tf.data.experimental.AUTOTUNE
BATCH_SIZE = 16 * strategy.num_replicas_in_sync
EPOCHS = 2
DATA_PATH =  "../input/lastjigasw"
SEQUENCE_LENGTH = 192
MODEL = 'jplu/tf-xlm-roberta-large'
OUTPUT_PATH = "/kaggle/working"
```

Tout comme dans le précédent modèle, on renvoie le tokenizer associé au modèle courant.

Paramètres : MODEL : str
Nom du modèle HuggingFace

Retour : tokenizer : Objet XLMRobertaTokenizer
Tokenizer XLM-Roberta

```
def get_tokenizer(MODEL):
    """Obtenez le tokenizer pour XLM-Roberta"""
    tokenizer = AutoTokenizer.from_pretrained(MODEL)

    return tokenizer

tokenizer = get_tokenizer(MODEL)
```

La fonction ***encode*** renvoie la représentation vectorielle(input_ids) de chaque token dans son contexte. Cette fonction prend en entrée l'ensemble du texte, la taille maximale de la séquence à encoder et enfin le *tokenizer* puis on appelle la fonction *batch_encode_plus* ou *encode_plus*.

Paramètres : texts : list
Liste des textes à encoder

maxlen : int
Longueur maximale de la séquence encodée

tokenizer : Objet XLMRobertaTokenizer
Tokenizer XLM-Roberta

Retour : input_ids : np.array
Liste contenant ids des tokens

```
def encode(texts, tokenizer, maxlen=512):
    enc_di = tokenizer.batch_encode_plus(
        texts,
        return_attention_masks=False,
        return_token_type_ids=False,
        pad_to_max_length=True,
        max_length=maxlen
    )
    return np.array(enc_di['input_ids'])
```

Utilisation de l'api ***tf.data.Dataset*** qui permet de créer un dataset à partir des données d'entrée, appliquer des transformations pour pré-traiter les données puis itérer sur l'ensemble des éléments. Elle permet une utilisation efficace(mémoire, temps d'exécution, etc) des entrées dans le pipeline.

```
train_dataset = (
    tf.data.Dataset
    .from_tensor_slices((x_train, y_train))
    .repeat()
    .shuffle(2048)
    .batch(BATCH_SIZE)
    .prefetch(AUTO)
)

valid_dataset = (
    tf.data.Dataset
    .from_tensor_slices((x_valid, y_valid))
    .batch(BATCH_SIZE)
    .cache()
    .prefetch(AUTO)
)

test_dataset = (
    tf.data.Dataset
    .from_tensor_slices(x_test)
    .batch(BATCH_SIZE)
)
```

La fonction ***build_model*** construit et renvoie le modèle à partir de l'architecture *transformer* de *XLM-Roberta*.

Nous avons rajouté à la sortie des couches du *transformer* une couche *Dropout* et une sortie binaire(lié au problème). Nous configurons les paramètres de la fonction *compile*.

Paramètres : transformer : TFRobertaModel

Couche architecture *transformer*

max_len : int

Utiliser pour la forme des entrées du modèle.

Retour : model : tf.keras.Model

Modèle construit

```
def build_model(transformer, max_len=512):
    """
    Construire le modèle à partir des couches Roberta
    """
    input_word_ids = Input(shape=(max_len,), dtype=tf.int32, name="input_word_ids")

    sequence_output = transformer(input_word_ids)[0]
    cls_token = sequence_output[:, 0, :]
    x = tf.keras.layers.Dropout(0.5)(cls_token)
    out = Dense(1, activation='sigmoid')(x)

    model = Model(inputs=input_word_ids, outputs=out)
    model.compile(tf.keras.optimizers.Adam(lr=1e-5), loss='binary_crossentropy', metrics=[tf.ke
ras.metrics.AUC()])

    return model
```

Chargement du modèle sur ***TPU***. la méthode *from_pretrained*de la ***TFAutoModel*** prend entrée un modèle et renvoie les couches pré-entraînées de ce dernier.

```
%%time
with strategy.scope():
    transformer_layer = TFAutoModel.from_pretrained(MODEL)
    model = build_model(transformer_layer, max_len=SEQUENCE_LENGTH)
```

Tout d'abord, nous entraînons notre modèle sur le dataset d'entrainement qui est entièrement en anglais.

```
n_steps = x_train.shape[0] // BATCH_SIZE
model.fit(train_dataset,steps_per_epoch=n_steps, validation_data=valid_dataset, epochs=EPOCHS)
```

Puis, nous l'entraînons sur l'ensemble de validation, qui est beaucoup plus petit mais contient un mélange de différentes langues.

```
n_steps2 = x_valid.shape[0] // BATCH_SIZE
model.fit(valid_dataset.repeat(), steps_per_epoch=n_steps2, epochs=EPOCHS)
```

Enfin, nous avons un exemple de code standard et commenté pour configurer les TPUs.

TPU Configs

```
# Detect hardware, return appropriate distribution strategy
try:
    # TPU detection. No parameters necessary if TPU_NAME environment variable is
    # set: this is always the case on Kaggle.
    tpu = tf.distribute.cluster_resolver.TPUClusterResolver()
    print('Running on TPU ', tpu.master())
except ValueError:
    tpu = None

if tpu:
    tf.config.experimental_connect_to_cluster(tpu)
    tf.tpu.experimental.initialize_tpu_system(tpu)
    strategy = tf.distribute.experimental.TPUStrategy(tpu)
else:
    # Default distribution strategy in Tensorflow. Works on CPU and single GPU.
    strategy = tf.distribute.get_strategy()

print("REPLICAS: ", strategy.num_replicas_in_sync)
```