Ray라는 모듈을 사용하여 하이퍼파라미터 서치를 하는 방법을 정리해봤습니다.
Ray tune은 빠르게 하이퍼파라미터 피팅을 할 수 있는 여러 알고리즘을 설정할 수 있고, 케라스 model train과 코드가 비슷해 빠르게 적응하실 수 있을 것입니다.

사실은 어제 밤부터 여러 시행착오를 겪었습니다..ㅜ 제가 쓴 글이 도움이 되었으면 하네요

(클라우드 환경에 이미 설치되어 있어 바로 사용할 수 있습니다.)

Project with Ray

데이터 전처리

```
import glob
import os, re
import numpy as np
import tensorflow as tf

txt_file_path = os.getenv('HOME')+'/aiffel/lyricist/data/lyrics/*'

txt_list = glob.glob(txt_file_path)

raw_corpus = []

# 여러개의 txt 파일을 모두 읽어서 raw_corpus 에 담습니다.
for txt_file in txt_list:
    with open(txt_file, "r") as f:
        raw = f.read().splitlines()
        raw_corpus.extend(raw)

>>> print("데이터 크기:", len(raw_corpus))
```

```
데이터 크기: 187088
```

데이터를 로드합니다

```
# 입력된 문장을
#     0. 괄호 안에 있는 내용을 없앱니다
#     1. 소문자로 바꾸고, 양쪽 공백을 지웁니다
#     2. 특수문자 양쪽에 공백을 넣고
#     2-1. '뒤에 공백 하나를 넣습니다
#     3. 여러개의 공백은 하나의 공백으로 바꿉니다
#     4. a-zA-Z?.!,¿,'가 아닌 모든 문자를 하나의 공백으로 바꿉니다
#     5. 다시 양쪽 공백을 지웁니다
#     6. (' )로 시작하는 경우 지웁니다
#     7. '앞에 공백이 있는 경우 함께 지웁니다.
#     8. 문장 시작에는 <start>, 끝에는 <end>를 추가합니다
# 이 순서로 처리해주면 문제가 되는 상황을 방지할 수 있겠네요!
def preprocess_lyrics(sentence):
    sentence = re.sub(r'\([^)]*\)',r'',sentence)
```

```
    sentence = re.sub(r'\[[^)]*\]',r'',sentence )
    sentence = sentence.lower().strip() # 1
    sentence = re.sub(r"([?.!,¿])", r" \1 ", sentence) # 2
    sentence = re.sub(r"([\'])", r"\1 ", sentence)  # 2-1
    sentence = re.sub(r'[" "]+', " ", sentence) # 3
    sentence = re.sub(r"[^a-zA-Z?.!,¿\']+", " ", sentence) # 4
    sentence = sentence.strip() # 5
    sentence = re.sub(r"^\' ", r'', sentence)  # 6
    sentence = re.sub(r" \'", r"", sentence)  # 7
    if sentence:
        sentence = '<start> ' + sentence + ' <end>' # 8
        return sentence

# 이 문장이 어떻게 필터링되는지 확인해 보세요.
>>> print(preprocess_lyrics(""))
```

데이터 전처리를 합니다. 저는 어퍼스트로피를 살리는 방식으로 진행했습니다. 만약 전처리 후 길이가 0인 문장이 되었다면 None이 반환되는데, 이는 아래서 말뭉치를 만드는 셀에서 해결합니다.

```
None
```

```
# 여기에 정제된 문장을 모을겁니다
lyric_corpus = []

for lyric in raw_corpus:
    # 정제를 합니다
    preprocessed_lyric = preprocess_lyrics(lyric)
    # 우리가 원하지 않는 문장은 건너뜁니다
    if not preprocessed_lyric: continue  # Nonetype이면 아무 행동없이 넘어갑니다
    lyric_corpus.append(preprocessed_lyric)
```

```
# 토큰화 할 때 텐서플로우의 Tokenizer와 pad_sequences를 사용합니다
# 더 잘 알기 위해 아래 문서들을 참고하면 좋습니다
#
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer
#
https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/pad_sequences
def tokenize(corpus):
    # 20000단어를 기억할 수 있는 tokenizer를 만들겁니다
    # 우리는 이미 문장을 정제했으니 filters가 필요없어요
    # 20000단어에 포함되지 못한 단어는 '<unk>'로 바꿀거에요
    tokenizer = tf.keras.preprocessing.text.Tokenizer(
        num_words=20000,
        filters=' ',
        oov_token="<unk>"
    )
    # corpus를 이용해 tokenizer 내부의 단어장을 완성합니다
    tokenizer.fit_on_texts(corpus)
    # 준비한 tokenizer를 이용해 corpus를 Tensor로 변환합니다
    tensor = tokenizer.texts_to_sequences(corpus)
    # 입력 데이터의 시퀀스 길이를 일정하게 맞춰줍니다
```

```
    # 만약 시퀀스가 짧다면 문장 뒤에 패딩을 붙여 길이를 맞춰줍니다.
    # 문장 앞에 패딩을 붙여 길이를 맞추고 싶다면 padding='pre'를 사용합니다
    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
padding='post', maxlen=15, truncating='post')

    print(tensor,tokenizer)
    return tensor, tokenizer

>>> tensor, tokenizer = tokenize(lyric_corpus)
```

```
[[  2   6 190 ...   0   0   0]
 [  2   9 542 ...   0   0   0]
 [  2  50  38 ...   0   0   0]
 ...
 [  2 310 153 ...   0   0   0]
 [  2  10 163 ...   0   0   0]
 [  2 177  15 ...   0   0   0]] <keras_preprocessing.text.Tokenizer object at
0x7f598b402410>
```

tokenize할 때, 사전 사이즈는 20000, maxlen=15, truncating='post'를 선택했습니다. 'post'를 선택한 이유는 길이가 15보다 긴 문장을 필터링하기 위해서입니다. 바로 다음 셀에서 진행됩니다.

```
# tensor에서 마지막 토큰을 잘라내서 소스 문장을 생성합니다
# 마지막 토큰은 <end>가 아니라 <pad>일 가능성이 높습니다.
# 마지막 토큰이 0이나 3이 아니라면 maxlen보다 긴 것입니다. (truncating = 'post')
# 따라서 마지막 토큰이 0이거나 3인것만 선택합니다.
short_tensor = np.array([x for i, x in  enumerate(tensor) if x[-1]==0 or
x[-1]==3])
>>> print(short_tensor.shape)
src_input = short_tensor[:,:-1]
# tensor에서 <start>를 잘라내서 타겟 문장을 생성합니다.
tgt_input = short_tensor[:,1:]

>>> print(src_input[0])
>>> print(tgt_input[0])
```

```
(154437, 15)
[  2   6 190   7 899   4 101   6  72  50   3   0   0   0]
[  6 190   7 899   4 101   6  72  50   3   0   0   0   0]
```

truncating ='post'를 선택한다면, 시퀀스의 뒷부분을 잘라내게(가지치기) 됩니다. 이때 15보다 긴 길이의 시퀀스는 문장 도중에 잘리게 됩니다. 이러한 경우의 마지막 자리는 이나 토큰이 아니게 될 것이므로, 그에 해당하는 조건을 세워 필터링을 합니다.

```
from sklearn.model_selection import train_test_split


enc_train, enc_val, dec_train, dec_val = train_test_split(src_input, tgt_input,
test_size=0.2, random_state=36)
>>> print(len(enc_train), len(enc_val))
```

데이터셋 분할에서 random_state를 고정했습니다. 이는 재현을 할 수 있도록 한 것입니다.

```
123549 30888
```

Model, Dataset 생성함수 제작

ray tutorial을 따라서 진행하였습니다.

```
class TextGenerator(tf.keras.Model):
    def __init__(self, vocab_size, embedding_size, hidden_size):
        super().__init__()

        self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_size)
        self.rnn_1 = tf.keras.layers.LSTM(hidden_size, return_sequences=True)
        self.rnn_2 = tf.keras.layers.LSTM(hidden_size, return_sequences=True)
        self.linear = tf.keras.layers.Dense(vocab_size)

    def call(self, x):
        out = self.embedding(x)
        out = self.rnn_1(out)
        out = self.rnn_2(out)
        out = self.linear(out)

        return out
```

```
def create_model(*, embedding_size, hidden_size, lr,
vocab_size=tokenizer.num_words+1):
    '''This is a model generating function so that we can search over neural net

    parameters and architecture'''

    optimizer = tf.keras.optimizers.Adam(lr)
    loss = tf.keras.losses.SparseCategoricalCrossentropy(
        from_logits=True,
    reduction='none'
    )

    model = TextGenerator(vocab_size, embedding_size, hidden_size)
    model.compile(loss=loss, optimizer=optimizer, metrics=['accuracy'])
    return model
```

tokenizer는 이미 선언되어 있다고 가정하고, TextGenerator 객체를 생성하는 createmodel을 정의합니다.

```
def create_dataset(enc, dec, *, batch_size=128, seed=36):
    buffer_size = len(enc)
    steps_per_epoch = len(enc) // batch_size

    # 준비한 데이터 소스로부터 데이터셋을 만듭니다
    # 데이터셋에 대해서는 아래 문서를 참고하세요
    # 자세히 알아둘수록 도움이 많이 되는 중요한 문서입니다
    # https://www.tensorflow.org/api_docs/python/tf/data/Dataset
    dataset = tf.data.Dataset.from_tensor_slices((enc, dec))
    dataset = dataset.shuffle(buffer_size, seed=seed)
    dataset = dataset.batch(batch_size, drop_remainder=True)
    return dataset
```

dataset도 만들어 줍니다.

Ray Tune를 사용해보자

본격적으로 시작입니다. Ray Tune Tutorial-Keras 를 변형하여 진행하였습니다.

```
!pip install 'Ray[tune]'
```

주피터 환경에서 실행해보면 이미 설치되어 있는 것을 확인할 수 있습니다.

```
class TuneReporterCallback(tf.keras.callbacks.Callback):
    """Tune Callback for Keras.

    The callback is invoked every epoch.
    """

    def __init__(self, logs={}):
        self.iteration = 0
        super(TuneReporterCallback, self).__init__()

    def on_epoch_end(self, batch, logs={}):
        self.iteration += 1
        tune.report(keras_info=logs, mean_accuracy=logs.get("accuracy"),
mean_loss=logs.get("loss"), val_loss=logs.get("val_loss"))
```

TuneReporterCallback이라는 Callback 클래스를 만들고 매 epoch 마다 keras log에서 원하는 정보를 가져옵니다.

중요: 최적화를 원하는 변수 var_name에 대해(이 경우 val_loss) tune.report(some_var=logs.get("var_name"))을 여기서 진행해주어야 실제 run에 metric인자로 줄 수 있습니다

```
from ray import tune
import pandas as pd
```

```
def tune_model(config):
    dataset_128 = create_dataset(enc_train, dec_train)
    dataset_val = create_dataset(enc_val, dec_val)
    # Hyperparameters
    embedding_size, hidden_size, lr = config["embedding_size"],
config["hidden_size"], config["lr"]
    model = create_model(embedding_size=embedding_size, hidden_size=hidden_size,
lr=lr)

    # Enable Tune to make intermediate decisions by using a Tune Callback hook.
This is Keras specific.
    callbacks =
[tf.keras.callbacks.ModelCheckpoint(filepath='best_model_ray.h5',
                                            monitor='val_loss',
                                            save_best_only=True),
                TuneReporterCallback()]

    # Train the model
    model.fit(dataset_128, validation_data=dataset_val, epochs=10,
callbacks=callbacks)
```

학습에 사용할 모델 생성함수를 만듭니다. create_dataset()과 create_model()이 여기에서 사용됩니다. tune_model()의 인자 `config`는 하이퍼파라미터를 찾을 영역(= search space)를 지정해줍니다.

```
def tune_exp_model(config):
    dataset_exp_5 = create_dataset(enc_train[:1000], dec_train[:1000],
batch_size=50)
    dataset_exp_val = create_dataset(enc_val[:250], dec_val[:250],
batch_size=10)
    # Hyperparameters
    embedding_size, hidden_size, lr = config["embedding_size"],
config["hidden_size"], config["lr"]
    model = create_model(embedding_size=embedding_size, hidden_size=hidden_size,
lr=lr)

    # Enable Tune to make intermediate decisions by using a Tune Callback hook.
This is Keras specific.
    callbacks =
[tf.keras.callbacks.ModelCheckpoint(filepath='best_model_ray.h5',
                                            monitor='val_loss',
                                            save_best_only=True),
                TuneReporterCallback()]

    # Train the model
    model.fit(dataset_exp_5, validation_data=dataset_exp_val, epochs=10,
callbacks=callbacks)
```

데이터셋의 크기가 크기 때문에 실험을 위해서는 작은 데이터셋을 이용하는 모델도 필요합니다. training _set은 1000개의 샘플, validation_set은 250개로 진행했습니다. train_test_split()에서 random_state를 고정했으므로, 늘 같은 데이터가 담깁니다.

```
hyperparameter_space = {
    "lr": tune.uniform(0.0001, 0.03),
    "embedding_size": tune.randint(128, 512),
    "hidden_size": tune.randint(192, 1536),
    "num_gpus": 1
}
```

lr, embedding_size, hidden_size 에 대한 search space를 설정합니다.

중요: "num_gpus": 1 을 설정하지 않는다면 gpu를 사용하지 않습니다

```
num_samples = 30
```

몇 번의 서치를 진행할지 정합니다.

```
import ray


ray.__version__
```

```
'1.1.0'
```

```
import numpy as np; np.random.seed(5)
from ray.tune import JupyterNotebookReporter


RAY_USE_MULTIPROCESSING_CPU_COUNT=1
reporter = JupyterNotebookReporter(overwrite=True)
reporter.add_metric_column("val_loss")


ray.shutdown()  # Restart Ray defensively in case the ray connection is lost.
ray.init(log_to_driver=False)
# We clean out the logs before running for a clean visualization later.
! rm -rf ~/aiffel/OOOProjects/EX4_data/ray_results


analysis = ray.tune.run(
    tune_model,
    metric="val_loss",
    verbose=2,
    config=hyperparameter_space,
    num_samples=num_samples,
    resources_per_trial={"cpu": 4, "gpu": 1},
    progress_reporter=reporter,
    local_dir="~/aiffel/OOOProjects/EX4_data/ray_results"
    )
```

```
assert len(analysis.trials) == num_samples, "Did you set the correct number of samples?"
```

최적화가 실행되는 셀입니다.

reporter.add_metric_column()의 인자로 metric을 넣어주면 됩니다. 이렇게 하게 되면 progress를 표시할 때, val_loss라는 열이 생기게 됩니다.

local_dir은 결과를 저장할 경로를 지정합니다.

이 셀을 실행하게 되면..

== Status ==
Memory usage on this node: 4.5/54.9 GiB
Using FIFO scheduling algorithm.
Resources requested: 4/4 CPUs, 1/1 GPUs, 0.0/26.12 GiB heap, 0.0/8.98 GiB objects (0/1.0 accelerator_type:K80)
Result logdir: /aiffel/aiffel/000Projects/EX4_data/ray_results/tune_model_2021-07-24_08-05-48
Number of trials: 9/30 (1 PENDING, 1 RUNNING, 7 TERMINATED)

Trial name	status	loc	embedding_size	hidden_size	lr	acc	loss	iter	total time (s)	val_loss
tune_model_f484f_00007	RUNNING	10.244.124.38:1681	265	233	0.00895279	0.51176	3.27786	1	97.4232	2.99271
tune_model_f484f_00008	PENDING		338	295	0.0148149					
tune_model_f484f_00000	TERMINATED		317	1190	0.0067376	0.631644	1.8876	10	3355.52	2.68697
tune_model_f484f_00001	TERMINATED		201	1224	0.0109757	0.520152	3.33231	10	3373.93	3.53315
tune_model_f484f_00002	TERMINATED		240	862	0.00278565	0.725689	1.36365	10	2351.93	2.45447
tune_model_f484f_00003	TERMINATED		332	1329	0.0146505	0.523726	2.95337	10	3846.13	3.11232
tune_model_f484f_00004	TERMINATED		208	219	0.00251416	0.581024	2.32788	10	857.804	2.65978
tune_model_f484f_00005	TERMINATED		331	1281	0.0132951	0.519762	3.18524	10	3705.9	3.37985
tune_model_f484f_00006	TERMINATED		214	338	0.0264101	0.529034	2.85434	10	1083.03	3.06126

이러한 결과를 얻을 수 있습니다. **이 결과를 얻는 데에는 기민퍼실님이 큰 도움을 주셨습니다**

ray.tune은 다양한 서치 알고리즘과 스케쥴러를 사용합니다. Search Algorithms. search_alg를 None으로 주면 기본 서치 알고리즘을 사용하게 됩니다.

```
best_config = analysis.get_best_config(metric="val_loss", mode='min')
logger.info(f'Best config: {best_config}')
```