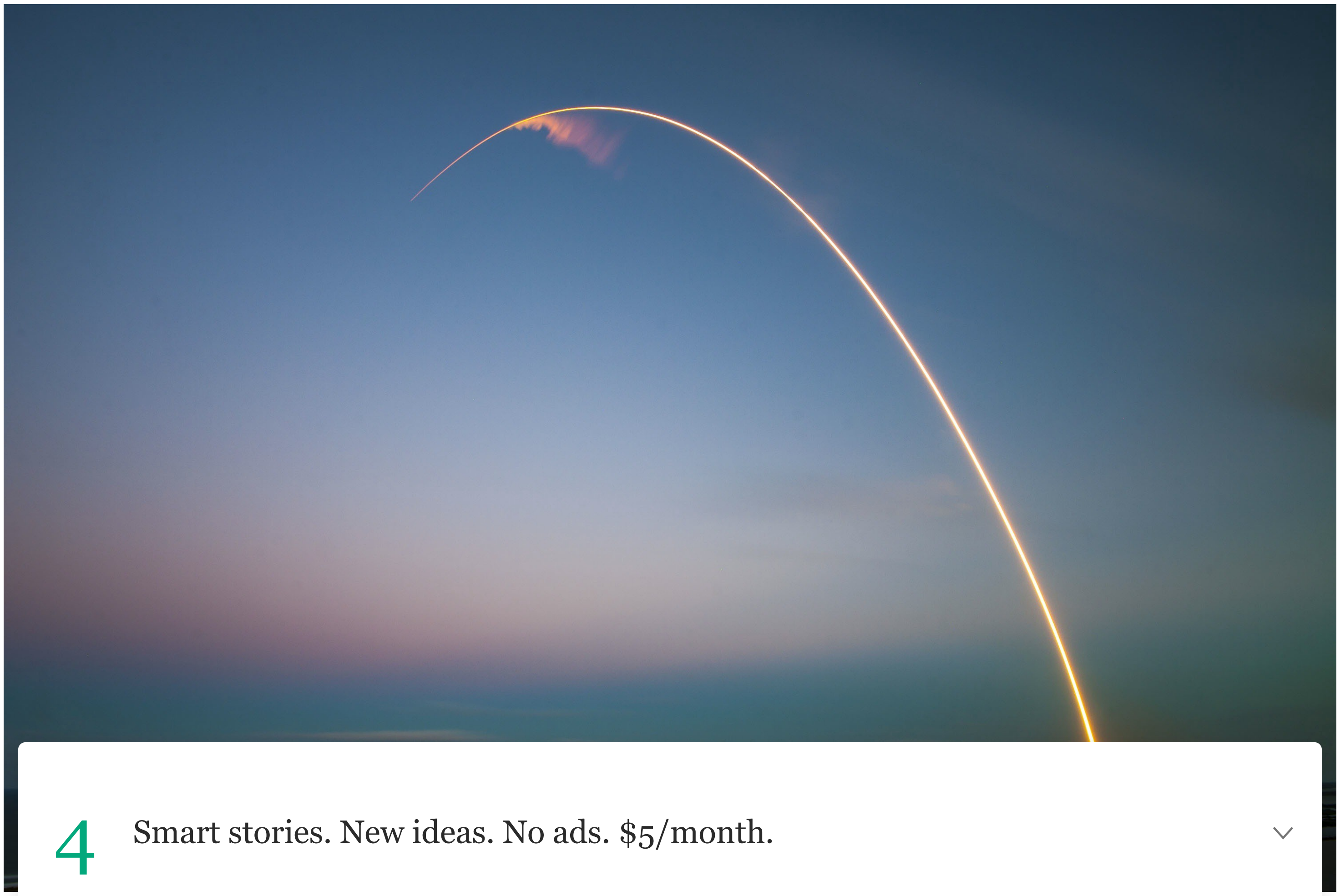
Getting Your Hands Dirty with TensorFlow 2.0 and Keras API

Diving into technical details of the regression model creation with TensorFlow 2.0 and Keras API. In TensorFlow 2.0, Keras comes out of the box with TensorFlow library. API is simplified and more convenient to use.

Andrej Baranovskij Follow
Nov 1 · 3 min read ★



Source: Pixabay

TensorFlow 2.0 comes with Keras packaged inside, there is no need to import Keras as a separate module (although you can do this if you need). TensorFlow 2.0 API is simplified and improved. This is good news for us — Machine Learning developers.

This is how you import Keras now, from TensorFlow:

```
from tensorflow import feature_column
from tensorflow import keras
from tensorflow.keras import layers

print(tf.__version__)
2.0.0
```

I'm using tf.data input pipeline to encode categorical columns, Keras API works well with tf.data. One of the main advantages of tf.data is the fact that it acts as a bridge between data and model. There is no need to transform data by yourself, enough to define transformation rule — transformed data automatically will be applied during training.

Data is fetched from CSV file into Pandas dataframe:

```
column_names = ['report_id','report_params','day_part','exec_time']
raw_dataframe = pd.read_csv('report_exec_times.csv')
dataframe = raw_dataframe.copy()

dataframe.head()
```

Column values for *report_params* vary, we need to normalize this column (make values to be on a similar scale):

I'm using a utility method (this method is taken from TensorFlow tutorial) to create tf.data dataset from Pandas dataframe:

```
def df_to_dataset(dataframe, shuffle=True, batch_size=32):
  dataframe = dataframe.copy()
  labels = dataframe.pop('exec_time')
  ds = tf.data.Dataset.from_tensor_slices((dict(dataframe), labels))
  if shuffle:
    ds = ds.shuffle(buffer_size=len(dataframe))
  ds = ds.batch(batch_size)
  return ds
```

Next, we need to define data mapping for categorical columns encoding. I'm using TensorFlow vocabulary list function, including mapping for all unique values (if there are many values, it would be better to use embedding API). Two columns are encoded — *report_id* and *day_part*:

```
feature_columns = []

feature_columns.append(feature_column.numeric_column('report_params')
)

report_id =
feature_column.categorical_column_with_vocabulary_list('report_id',
['1', '2', '3', '4', '5'])
report_id_one_hot = feature_column.indicator_column(report_id)
feature_columns.append(report_id_one_hot)

day_part =
feature_column.categorical_column_with_vocabulary_list('day_part',
['1', '2', '3'])
day_part_one_hot = feature_column.indicator_column(day_part)
feature_columns.append(day_part_one_hot)
```

Create Keras dense features layer out of array with TensorFlow encodings. We will use this layer during Keras model construction to define model training features:

We are done with features. Next, convert Pandas dataframe to tf.data with the help of utility function:

```
batch_size = 32
train_ds = df_to_dataset(train, batch_size=batch_size)
val_ds = df_to_dataset(val, shuffle=False, batch_size=batch_size)
test_ds = df_to_dataset(test, shuffle=False, batch_size=batch_size)
```

Dense features layer is used when Keras sequential model is defined (there is no need to pass array of features later into a *fit* function):

```
def build_model(feature_layer):
  model = keras.Sequential([
    feature_layer,
    layers.Dense(16, activation='relu'),
    layers.Dense(16, activation='relu'),
    layers.Dense(1)
  ])

  optimizer = keras.optimizers.RMSprop(0.001)

  model.compile(loss='mse',
                optimizer=optimizer,
                metrics=['mae', 'mse'])
  return model
```

Training is executed through *model.fit* function. We are using tf.data input pipeline to pass training and validation sets:

```
history = model.fit(train_ds,
              validation_data=val_ds,
              epochs=EPOCHS,
              callbacks=[early_stop])
```

The awesome thing about it — data encoding happens behind the scenes, based on the

Construct Pandas dataframe with input data:

```
headers = ['report_id', 'report_params', 'day_part']
dataframe_input = pd.DataFrame([[1, 15, 3]],
                                columns=headers,
                                dtype=float,
                                index=['input'])
```

Convert *report_params* value to be on the same scale as it was for the training:

```
eps=0.001 # 0 => 0.1¢
dataframe_input['report_params'] =
np.log(dataframe_input.pop('report_params')+eps)
```

Create tf.data input pipeline out of Pandas dataframe:

```
input_ds = tf.data.Dataset.from_tensor_slices(dict(dataframe_input))
input_ds = input_ds.batch(1)
```

Run *model.predict* function:

```
res = model.predict(input_ds)
print(res)
```

Resources:

- Source code with sample data is available on my GitHub repo

Enjoy !