Intro to Deep learning project - Using AI/ML for Energy Saving in 5G networks

Introduction

Energy costs for mobile networks can be as high as 10 percent of the total operating expenditure and it keeps increasing with every generation of cellular technology and ever-increasing demand for bandwidth. It is also estimated that radio access network (RAN) accounts for 73 percent of this cost. In addition, telecom operators are under significant pressure to reduce their carbon footprint as telecom already accounts for 2 to 3 percent of total global energy demand, as per a recent McKinsey report. Due to these reasons, mobile operators want to deploy intelligent solutions to optimize the energy consumption of their mobile networks.

Energy saving approaches

One of the approaches to saving energy in RAN is to turn off cells or frequency carriers which are not needed from a capacity point of view during low load hours (for example, from midnight to 6:00 am in a business district). Several solutions exist where traffic load is monitored and when it falls below a threshold, the cell is partially turned off. However, there's latency involved in waking up sleeping cells and there is also a need for customization of the load thresholds per cell, as the load characteristics among sites vary significantly. These issues can be addressed by AI/ML-based approaches where the future traffic load is predicted beforehand and cells are turned on/off accordingly.

Estimating future load and cell QoS correctly is essential for designing intelligent energy saving solutions. For example, if the predicted load is lower than the actual load, switching off a cell may cause an outage situation as the cell is switched off while its actual and immediate future load is high. On the other hand, in the case of load over prediction, cells may not be switched off when the actual future load is low resulting in less energy savings. Thus, there is an innate tradeoff between seeking maximum energy savings and minimization of service quality degradation. Furthermore, each cell in a network may have different load distribution characteristics, different energy saving priorities, and different degree of traffic imbalances, needing the ML training on a per cell level based on the corresponding data characteristics and operator preference.

SMaRT-5G

Open Networking Forum's (ONF) Sustainable Mobile and RAN Transformation 5G (SMaRT-5G) project is a collaborative effort to develop and demonstrate an ML-driven, intelligent energy savings solution for mobile networks. A Proof-of-Concept (PoC) implementation for an AI/ML based solution is designed and developed to experiment with various concepts for energy savings. Some of the design considerations are:

1. Power consumption as well as performance should be considered while determining the right AI/ML model to use for energy saving solutions. This can aid the selection among available sophisticated models such as neural networks, graph-based models, attention-based models.

2. The model training should accommodate bias in the network field data rather than assuming well-balanced loading among cells.

3. The model should be able to take input from the operator to prioritize energy saving vs cell QoS.

4. The model should take into account different distributions of data across different cells while coming up with an appropriate energy saving model. At one extreme, an AI/ML model could be trained on a per-cell basis, and at the other extreme, there could be one common model trained on data from thousands of cells. The right model training approach should be identified based on data availability, computing resources etc.

5. The model for final deployment should be trained and evaluated on real field data as the simulated traffic patterns may not capture sudden peaks, or valleys in the actual traffic. Simulated data could be used for developing a coarse model, initially.

6. The model should be easy to re-train based on load distribution shifts in the network.

7. Model results and predictions should be made available, so that they can be reused by other applications or services. Also, a real-time check of traffic shift is often necessary (and reacting accordingly) if a high weight is given to QoS.

The Proof-of-Concept (PoC) implementation is available on GitHub. It uses simulated data to train a 3-layer fully-connected deep neural network which achieves an Accuracy of 50% (for error rate less than 20%). A copy of the results is included in this report.

Recurrent neural networks (RNN)

Predicting traffic load is based on previous history and can benefit from a RNN model which is inherently designed for such sequential data. LSTM (Long short-term memory) is a variation of an RNN model. An RNN can only memorize short-term information, but LSTM can handle long time-series data. Moreover, LSTM can prevent RNN's vanishing gradient problem during training.

Code

```
#!/usr/bin/env python3
#SPDX-License-Identifier: Apache-2.0
#Copyright 2024 Intel Corporation
from flask import Flask,request
import pandas as pd
import tensorflow as tf
import numpy as np
import json
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Input, LSTM, Bidirectional, Dense, Embedding
from keras import layers
from keras.callbacks import ModelCheckpoint, EarlyStopping

model = None

def convert2matrix(data_arr, look_back, full_data):
X, Y =[], []
for i in range(len(data_arr)-look_back):
 d=i+look_back
 xdata = data_arr[i:d]
 X.append(xdata)
 Y.append(data_arr[d])
return np.array(X), np.array(Y)
def model_dnn(look_back):
   model = Sequential()
   model.add(Input(shape=(look_back), batch_size=10, dtype=tf.float32))
   model.add(layers.Embedding(input_dim=300, output_dim=64))
   model.add(LSTM(32))
   #model.add(LSTM(32, return_sequences=True))  # Bidirectional LSTM
   #model.add(LSTM(16))  # Additional LSTM layer
   model.add(Dense(look_back, activation='relu'))
   model.add(Dense(1))
   model.compile(loss= "mse",  optimizer='adam', metrics = ['mse', 'mae'])
   return model

df = pd.DataFrame(pd.read_csv("load_test.csv"))
train_size = 300
val_size = 84
train, val, test = (df['Load'].tolist())[0:train_size],
(df['Load'].tolist())[train_size:train_size+val_size],
(df['Load'].tolist())[train_size+val_size:len(df.values)]
look_back = 8
print("-----------------------------------------")
trainX, trainY = convert2matrix(train, look_back, df[0:train_size])
```

```
valX, valY = convert2matrix(val, look_back,
df[train_size:train_size+val_size])
testX, testY = convert2matrix(test, look_back, df[train_size+val_size:])
model = model_dnn(look_back)

model.fit(trainX,trainY, epochs=1000, batch_size=10, verbose=2,
validation_data=(valX,valY),
         callbacks=[EarlyStopping(monitor='val_loss',
patience=100)],shuffle=False)

print("Evaluate on test data")

# Reshape testX to have 3D shape (num_samples, look_back, 1)
tX = np.reshape(testX, (testX.shape[0], look_back, 1))
Z = model.predict(tX)

# Print predictions and actual values
for i in range(len(testY)):
   print(f"Predicted: {Z[i][0]:.6f}, \tActual: {testY[i]}, \t% Error:
{abs(Z[i][0]-testY[i])/testY[i]:.2f}")
```

Results

A basic LSTM model was trained with one input layer that defines its input type as floating point, one Embedding layer that turns the input into fixed length vectors and a LSTM layer with 32 units and a fully connected layer and an output layer. It took 30s to train the model in 134 epochs and achieved an Accuracy of 71% (for error rate less than 20%). Other architectures with additional LSTM layers and bidirectional LSTMs and different hyper parameters were also trained, but they didn't result in any significant improvements. It was difficult to find the accuracy of the model, as the training and target data are floating point values and predictions can only be approximate values. So, the built-in evaluate function didn't give any meaningful results and the test data had to predicted with the predict function and compared with the target value side by side.

Conclusion

LSTM models are inherently very good in inferring sequential data. A basic LSTM model was trained and proven to be an improvement over fully connected DNNs. The model took 30s to train, so it satisfies the ease of training requirement cited above. The other requirements were not considered, but they can be easily be added to this model.