E2E ASR

December 4, 2021

1 End-To-End Automatic Speech Recognition

The E2E models typically consist of one model that does all the work, in comparison to the conventional models that are composed of acoustic model, language model and lexicon. There are two main approaches for doing E2E ASR, the connectionist temporal classification (CTC) and attention-based encoder-decoder (AED).

In this exercise, we will see a simple example of doing E2E ASR using attention-based encoder-decoder architecture. The model is pre-trained on the LibriSpeech dataset, which contains recordings of about 1000 hours.

Throughout the exercise, you will test the performance of the model on various test sets and see how the perormance varies depending on the test set being used. Additionally, you will familiarize yourself with how the decoding is done and what are some advantages and drawbacks of it.

The answers should be uploaded to mycourses page in a PDF format.

1.1 Data preparation

The first thing that we need to do it to prepare the data . We will start by importing the necessary libraries. The model is developed using the Pytorch deep learning framework. More details about Pytorch can be found here.

[1]:
```
# The 'numpy' library contains functions for various vector and matrix␣
↪operations
import numpy as np

# jiwer is a library for calculating the WER
!pip install jiwer
from jiwer import wer

# 'torch' is the deep learning framework that we are going to use to develop,␣
↪train and test the model
import torch
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence,␣
↪pad_sequence
```

```
import torch.optim as optim

# 'prepare_data' contain various functions for preparing the data for training␣
 ↪and inference
import prepare_data
# 'train' contains the code used for training the model
from train import train
# 'calculate_wer' contains a script for calculating the word error rate (WER)
from calculate_wer import get_word_error_rate
```

```
Collecting jiwer
  Downloading jiwer-2.3.0-py3-none-any.whl (15 kB)
Collecting python-Levenshtein==0.12.2
  Downloading python-Levenshtein-0.12.2.tar.gz (50 kB)
     |                 | 50 kB 6.0 MB/s eta 0:00:011
Requirement already satisfied: setuptools in
/opt/conda/lib/python3.8/site-packages (from python-Levenshtein==0.12.2->jiwer)
(49.6.0.post20201009)
Building wheels for collected packages: python-Levenshtein
  Building wheel for python-Levenshtein (setup.py) … done
  Created wheel for python-Levenshtein:
filename=python_Levenshtein-0.12.2-cp38-cp38-linux_x86_64.whl size=145667
sha256=f21a4c887d1d11cdd47c75a54862075cfabdd19483a0181932894594af0fdaff
  Stored in directory: /home/yang2/.cache/pip/wheels/d7/0c/76/042b46eb0df65c3ccd
0338f791210c55ab79d209bcc269e2c7
Successfully built python-Levenshtein
Installing collected packages: python-Levenshtein, jiwer
Successfully installed jiwer-2.3.0 python-Levenshtein-0.12.2
```

To ensure that we get the same results every time we run the exercise, we can set a seed for generating random numbers, using the command `torch.manual_seed(0)`.

The Pytorch framework allows the computations to be done on a CPU or a GPU. The command `torch.device("cuda:0" if torch.cuda.is_available() else "cpu")` checks if a GPU with CUDA is installed, and if it is, it will run the computations on it, otherwise it will run everything on the CPU.

[3]:
```
torch.manual_seed(0)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
device
```

[3]:
```
device(type='cpu')
```

The cell below loads the test data that we will use to evaluate the model. For features, we will use log filter banks, with 40 filters. The targets consist of manually annotated transcripts.

1. The `test_clean` is a subset of the LibriSpeech dataset, that has utterances on which the system achieves lower WER.
2. The `test_other` is a subset of the LibriSpeech dataset, that has utterances on which the

system achieves higher WER.

3. The `test_long` is a subser of the LibriSpeech dataset, that has long utterances.

You can find more information about the dataset in the original paper

[5]:
```
features_test_clean = prepare_data.load_features('data/test_clean.npy')
target_test_clean = prepare_data.load_transcripts('data/test_clean.txt')

features_test_other = prepare_data.load_features('data/test_other.npy')
target_test_other = prepare_data.load_transcripts('data/test_other.txt')

features_test_long = prepare_data.load_features('data/test_long.npy')
target_test_long = prepare_data.load_transcripts('data/test_long.txt')
```

With the cell below, we can inspect the shape of the features of the first data point.

[6]:
```
features_test_clean[0].size()
```

[6]:
```
torch.Size([1042, 40])
```

In this case, the first value (1042) is the number of frames and the second value (40) is the number of filters.

Since we will be using a pre-trained model, we don't need train and development sets. In case some students want to see how the training runs, we will assign the training and development sets to be same as the test set. That way, there will be some training data to play with.

[7]:
```
features_train = features_test_clean
target_train = target_test_clean
features_dev = features_test_clean
target_dev = target_test_clean
```

Next, we want to create dictionaries that map each character to an index and vice versa. The whole character set consists of all the lower-case characters, plus empty space , `'`, and the special tokens `<sos>` and `<eos>`, indicating the start and the end of each sample.

[8]:
```
char2idx, idx2char = prepare_data.encode_data()
print(char2idx)
```

```
{'<sos>': 1, '<eos>': 2, 'a': 3, 'b': 4, 'c': 5, 'd': 6, 'e': 7, 'f': 8, 'g': 9,
'h': 10, 'i': 11, 'j': 12, 'k': 13, 'l': 14, 'm': 15, 'n': 16, 'o': 17, 'p': 18,
'q': 19, 'r': 20, 's': 21, 't': 22, 'u': 23, 'v': 24, 'w': 25, 'x': 26, 'y': 27,
'z': 28, "'": 29, ' ': 30}
```

Using the cell below, we can replace each character in the transcripts with the appropriate index.

[9]:
```
# convert labels to indices
indexed_target_train = prepare_data.label_to_idx(target_train, char2idx)
indexed_target_dev = prepare_data.label_to_idx(target_dev, char2idx)
```

```
indexed_target_test_clean = prepare_data.label_to_idx(target_test_clean,␣
 ↪char2idx)
indexed_target_test_other = prepare_data.label_to_idx(target_test_other,␣
 ↪char2idx)
indexed_target_test_long = prepare_data.label_to_idx(target_test_long, char2idx)
```

Next, we will group the features and the indexed targets in a tuple.

```
[10]: # combine features and labels in a tuple
train_data = prepare_data.combine_data(features_train, indexed_target_train)
dev_data = prepare_data.combine_data(features_dev, indexed_target_dev)
test_clean_data = prepare_data.combine_data(features_test_clean,␣
 ↪indexed_target_test_clean)
test_other_data = prepare_data.combine_data(features_test_other,␣
 ↪indexed_target_test_other)
test_long_data = prepare_data.combine_data(features_test_long,␣
 ↪indexed_target_test_long)
```

The next cell divides the data in equal batches that are used for training.

```
[11]: batch_size = 10

pairs_batch_train = DataLoader(dataset=train_data,
                    batch_size=batch_size,
                    drop_last=True,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)

pairs_batch_dev = DataLoader(dataset=dev_data,
                    batch_size=batch_size,
                    drop_last=True,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)
```

1.2 Building the model

For building the model, we will use attention-based encoder-decoder architecture. For more details about the encoder-decoder architecture, refer to this Pytorch tutorial.

In the figure below, you can see an illustration of the attention-based encoder-decoder architecture. The figure is borrowed from this blog post.

The architecture consist of encoder and attention decoder. The encoder is a BLSTM that takes the audio features as input and outputs a vector representation of those features. The encoder is defined in the cell below:

[12]:
```
class Encoder(nn.Module):
    def __init__(self, input_tensor, hidden_size, num_layers):
        super(Encoder, self).__init__()

        self.input_tensor = input_tensor
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # The dropout randomly disconnects neurons during training. It is used␣
↪to prevent overfitting
        self.dropout = nn.Dropout(0.1)

        # The definition of the BLSTM cell, which takes the audio features,␣
↪processes them and returns a vector representation of them
        self.lstm = nn.LSTM(
                            self.input_tensor,
                            self.hidden_size,
                            num_layers=self.num_layers,
                            bidirectional=True
                            )


    def forward(self, input_tensor, input_feature_lengths):
        input_tensor = pack_padded_sequence(input_tensor, input_feature_lengths)
        output, hidden = self.lstm(input_tensor)
        output = pad_packed_sequence(output)[0]
        output = output[:, :, :self.hidden_size] + output[:, :, self.
↪hidden_size:]
        output = self.dropout(output)

        return output, hidden
```

The decoder consists of LSTM and attention mechanism. It is initialized using the hidden states of the encoder and uses the vector representation from the encoder to predict the next character, conditioned on the previous predictions. For the attention mechanism, we will use hybrid + location-aware attention, explained in more detail here. The decoder is defined in the cell below:

[13]:
```
class Decoder(nn.Module):
    def __init__(self, embedding_dim, encoder_hidden_size,␣
↪attention_hidden_size, output_size, num_layers, encoder_num_layers,␣
↪num_filters, batch_size, device):
        super(Decoder, self).__init__()

        self.encoder_hidden_size = encoder_hidden_size
        self.attention_hidden_size = attention_hidden_size
        self.num_filters = num_filters
        self.output_size = output_size
        self.embedding_dim = embedding_dim
        self.num_layers = num_layers
        self.encoder_num_layers = encoder_num_layers
        self.batch_size = batch_size
        self.dropout = nn.Dropout(0.1)
        self.device = device

        # the embedding transforms the characters to vector representations
        self.embedding = nn.Embedding(output_size, embedding_dim)

        self.lstm = nn.LSTM(self.embedding_dim,
                            self.encoder_hidden_size,
                            num_layers=self.num_layers,
                            bidirectional=False)
        self.out = nn.Linear(self.encoder_hidden_size*2, self.output_size)


        # initialization of the parameters needed for the attention calculation
        self.v = nn.Parameter(torch.FloatTensor(1, self.encoder_hidden_size).
↪uniform_(-0.1, 0.1))
        self.b = nn.Parameter(torch.FloatTensor(self.encoder_hidden_size).
↪uniform_(-0.1, 0.1))
        self.W_1 = torch.nn.Linear(self.encoder_hidden_size, self.
↪attention_hidden_size, bias=False)
        self.W_2 = torch.nn.Linear(self.encoder_hidden_size, self.
↪attention_hidden_size, bias=False)
        self.W_3 = nn.Linear(self.num_filters, self.attention_hidden_size,␣
↪bias=False)
        self.conv = nn.Conv1d(in_channels=1, out_channels=self.num_filters,␣
↪kernel_size=3, padding=1)

    def forward(self, input_tensor, decoder_hidden, encoder_output,␣
↪attn_weights):
        embedding = self.embedding(input_tensor)
        embedding = embedding.permute(1, 0, 2)

        # we pass through the LSTM the embedding of the character and␣
↪initialize the LSTM with the hidden state of the encoder
        decoder_output, decoder_hidden = self.lstm(embedding, decoder_hidden)


         # --------- calculation of the attention ---------
        try:
            conv_feat = self.conv(attn_weights).permute(0, 2, 1)
        except:
            random_tensor = torch.rand(encoder_output.size(1), 1,␣
↪encoder_output.size(0)).to(self.device)
            conv_feat = self.conv(F.softmax(random_tensor, dim=-1)).to(self.
↪device).permute(0, 2, 1)

        conv_feat = conv_feat.permute(1, 0, 2)
        scores = self.hybrid_attention_score(encoder_output, decoder_output,␣
↪conv_feat)
        scores = scores.permute(1, 0, 2)
        attn_weights = F.softmax(scores, dim=0)

        context = torch.bmm(attn_weights.permute(1, 2, 0), encoder_output.
↪permute(1, 0, 2))
        context = context.permute(1, 0, 2)
        output = torch.cat((context, decoder_output), -1)
        # --------- end of the attention calculation ---------


        output = self.out(output[0])
        output = self.dropout(output)
        output = F.log_softmax(output, 1)

        return output, decoder_hidden, attn_weights


    def hybrid_attention_score(self, encoder_output, decoder_output, conv_feat):
        out = torch.tanh(self.W_1(decoder_output) + self.W_2(encoder_output) +␣
↪self.W_3(conv_feat) + self.b)
        v = self.v.repeat(encoder_output.data.shape[1], 1).unsqueeze(1)
        out = out.permute(1, 0, 2)
        v = v.permute(0, 2, 1)
        scores = out.bmm(v)

        return scores
```

In the cell below, we define the hyperparameters of the network:

[14]:
```
encoder_layers = 5
decoder_layers = 1

encoder_hidden_size = 150
attention_hidden_size = 150

embedding_dim_chars = 100
num_filters = 100

encoder_lr = 0.0005
```

```
decoder_lr = 0.0005

num_epochs = 10
MAX_LENGTH = 800
skip_training = True
```

Next, we are going to initialize the encoder, decoder and the optimizers:

[15]:
```
# initialize the Encoder
encoder = Encoder(features_train[0].size(1), encoder_hidden_size,␣
 ↪encoder_layers).to(device)
encoder_optimizer = optim.Adam(encoder.parameters(), lr=encoder_lr)

# initialize the Decoder
decoder = Decoder(embedding_dim_chars, encoder_hidden_size,␣
 ↪attention_hidden_size, len(char2idx)+1, decoder_layers, encoder_layers,␣
 ↪num_filters, batch_size, device).to(device)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=decoder_lr)
```

Now, count the number of trainable parameters:

[16]:
```
print(encoder)
print(decoder)

total_trainable_params_encoder = sum(p.numel() for p in encoder.parameters() if␣
 ↪p.requires_grad)
total_trainable_params_decoder = sum(p.numel() for p in decoder.parameters() if␣
 ↪p.requires_grad)

print('The number of trainable parameters is: %d' %␣
 ↪(total_trainable_params_encoder + total_trainable_params_decoder))
```

```
Encoder(
  (dropout): Dropout(p=0.1, inplace=False)
  (lstm): LSTM(40, 150, num_layers=5, bidirectional=True)
)
Decoder(
  (dropout): Dropout(p=0.1, inplace=False)
  (embedding): Embedding(31, 100)
  (lstm): LSTM(100, 150)
  (out): Linear(in_features=300, out_features=31, bias=True)
  (W_1): Linear(in_features=150, out_features=150, bias=False)
  (W_2): Linear(in_features=150, out_features=150, bias=False)
  (W_3): Linear(in_features=100, out_features=150, bias=False)
  (conv): Conv1d(1, 100, kernel_size=(3,), stride=(1,), padding=(1,))
)
The number of trainable parameters is: 2624331
```

1.3 Training

This section implements the trainng of the E2E model. As a loss function, we are going to use negative log-likelihood. The function `train()` does the training. Since training an E2E model requires a lot of time and computational power, we will skip the training and load a pre-trained model instead.

Although it is not necessary for this exercise, if you want to see how the training is done, you can set the variable `skip_training` to `False`. For testing purposes, the training and development sets are the same as the test set. In practice we need to have separate training and development sets.

[17]:
```
if skip_training == False:
    # The criterion is the loss function that we are going to use. In this case␣
 ↪it is the negative log-likelihood loss.
    criterion = nn.NLLLoss(ignore_index=0, reduction='mean')
    train(pairs_batch_train, pairs_batch_dev, encoder, decoder,␣
 ↪encoder_optimizer, decoder_optimizer, criterion, batch_size, num_epochs,␣
 ↪device)
else:
    # load the pre-trained model
    checkpoint = torch.load('weights/state_dict_10.pt', map_location=torch.
 ↪device('cpu'))
    encoder.load_state_dict(checkpoint['encoder'])
    decoder.load_state_dict(checkpoint['decoder'])
```

1.4 Inference

Next, we are going to test the model's performance. The function `greedy_decoding()` uses the trained model to generate transcripts based on audio features. The greedy decoding takes the output of the decoder, which is a probability distribution over all the characters, and picks the most probable one at each timestep. The prediction of the current character is conditioned on the previous predictions. You can familiarize yourself more with various types of decoding strategies here.

For assessing the performance, we are going to use the WER metric.

[18]:
```
def greedy_decoding(encoder, decoder, batch_size, idx2char, test_data,␣
 ↪MAX_LENGTH, print_predictions):
    print('Evaluating...')

    # set the encoder and the decoder to evaluation mode
    encoder.eval()
    decoder.eval()

    with torch.no_grad():
        all_predictions = []
        all_labels = []

        for l, batch in enumerate(test_data):
            pad_input_seqs, input_seq_lengths, pad_target_seqs,␣
↪pad_target_seqs_lengths = batch
            pad_input_seqs, pad_target_seqs = pad_input_seqs.to(device),␣
↪pad_target_seqs.to(device)

            # pass the data through the encoder
            encoder_output, encoder_hidden = encoder(pad_input_seqs,␣
↪input_seq_lengths)

            decoder_input = torch.ones(batch_size, 1).long().to(device)
            decoder_hidden = (encoder_hidden[0].sum(0, keepdim=True),␣
↪encoder_hidden[1].sum(0, keepdim=True))

            attn_weights = torch.nn.functional.softmax(torch.
↪ones(encoder_output.size(1), 1, encoder_output.size(0)), dim=-1).to(device)

            predictions = []
            true_labels = []

            # decoding
            for i in range(MAX_LENGTH):
                attn_weights = attn_weights.squeeze()
                attn_weights = attn_weights.unsqueeze(0)
                attn_weights = attn_weights.unsqueeze(0)
                decoder_output,  decoder_hidden, attn_weights =␣
↪decoder(decoder_input, decoder_hidden, encoder_output, attn_weights)
                _, topi = decoder_output.topk(1)
                decoder_input = topi.detach()

                # if we get `<eos>`, stop the decoding
                if topi.item() == 2:
                    break
                else:
                    predictions.append(topi)

            true_labels = pad_target_seqs

            predictions = [idx2char[c.item()] for c in predictions if c.item()␣
↪not in (1, 2)]
            true_labels = [idx2char[c.item()] for c in true_labels if c.item()␣
↪not in (1, 2)]

            predictions = ''.join(predictions)
            true_labels = ''.join(true_labels)

            if print_predictions == True:
                print('\n')
                print('True: ', true_labels)
                print('Pred: ', predictions)

            all_predictions.append(predictions)
            all_labels.append(true_labels)


        print('\n')
        print('Word error rate: ', wer(all_labels, all_predictions))
```

[19]:
```
batch_size = 1

pairs_batch_test_clean = DataLoader(dataset=test_clean_data,
                    batch_size=batch_size,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)

pairs_batch_test_other = DataLoader(dataset=test_other_data,
                    batch_size=batch_size,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)

pairs_batch_test_long = DataLoader(dataset=test_long_data,
                    batch_size=batch_size,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)
```

The variables `pairs_batch_test_clean`, `pairs_batch_test_other` and `pairs_batch_test_long` contain subsets of the LibriSpeech test set. With the command below, we can test the performance of the model on the data stored in `pairs_batch_test_clean`.

[20]:
```
print_predictions = False
greedy_decoding(encoder, decoder, batch_size, idx2char, pairs_batch_test_clean,␣
 ↪MAX_LENGTH, print_predictions)
```

```
Evaluating…


Word error rate:  0.23870417732310314
```

[21]:
```
print_predictions = False
greedy_decoding(encoder, decoder, batch_size, idx2char, pairs_batch_test_other,␣
 ↪MAX_LENGTH, print_predictions)
```

```
Evaluating…


Word error rate:  0.3730108211330363
```

By running the cell below, we can compare the predictions against the true labels for the first 10 samples of the dataset.

[22]:
```python
print_predictions = True
test_clean_subset = test_clean_data[:10]

pairs_batch_clean_subset = DataLoader(dataset=test_clean_subset,
                    batch_size=batch_size,
                    shuffle=False,
                    collate_fn=prepare_data.collate,
                    pin_memory=True)
greedy_decoding(encoder, decoder, batch_size, idx2char,␣
 ↪pairs_batch_clean_subset, MAX_LENGTH, print_predictions)
```

```
Evaluating…


True:  he hoped there would be stew for dinner turnips and carrots and bruised
potatoes and fat mutton pieces to be ladled out in thick peppered flour fattened
sauce
Pred:  he hoped there would be stoover dinner turnips and carriets and bruised
potatoes and fat mutton pieces to be lateled out and pieces to be lateled out
and thick peppered flower fatten sauce


True:  stuff it into you his belly counselled him
Pred:  stuffered into you his belly councled him


True:  after early nightfall the yellow lamps would light up here and there the
squalid quarter of the brothels
Pred:  after early nightfall the yellow lamps would light up here and there the
squallete quarter of the brauffles


True:  hello bertie any good in your mind
Pred:  hellow burty and he good in your mind


True:  number ten fresh nelly is waiting on you good night husband
Pred:  number then fresh nell he is waiting on you could not husband
```

```
True:  the music came nearer and he recalled the words the words of shelley's
fragment upon the moon wandering companionless pale for weariness
Pred:  the music came nearer and he recalled the words the words of shellies
fragment upon the moon wandering companionless pale for weariness


True:  the dull light fell more faintly upon the page whereon another equation
began to unfold itself slowly and to spread abroad its widening tail
Pred:  the dull light felmor faintly upon the page whereon another requation
began to one fold itself slowly and to spread abroad its widening tale


True:  a cold lucid indifference reigned in his soul
Pred:  a cold lucid indifference reigned in his soul


True:  the chaos in which his ardour extinguished itself was a cold indifferent
knowledge of himself
Pred:  the chaos in which his ardor extinguished itself was a cold indifferent
knowledge of himself


True:  at most by an alms given to a beggar whose blessing he fled from he might
hope wearily to win for himself some measure of actual grace
Pred:  at most by an arms given to a beggar whose blessing he fled from he might
hope wearily to wind for himself some measure of actual grace


Word error rate:  0.22023809523809523
```

1.4.1 Question 1:

1. Report the WER on the data stored in `pairs_batch_test_clean` and `pairs_batch_test_other`. Use up to two decimal points.

`pairs_batch_test_clean`: WER=0.24

`pairs_batch_test_other`: WER=0.37

2. Why do you think the WER is worse on `pairs_batch_test_other`? Which factors could impact the performance?

In "LIBRISPEECH: AN ASR CORPUS BASED ON PUBLIC DOMAIN AUDIO BOOKS" where the dataset come from, it introduces the difference of recording quality and accents vary test performances. The model was trained based on higger recording quality and US English accents. when the test dataset suffer the lower recording quality and various accents, The performance can't be better on the clean dataset.

[23]:
```
print_predictions = False
```

```
greedy_decoding(encoder, decoder, batch_size, idx2char, pairs_batch_test_long,␣
↪MAX_LENGTH, print_predictions)
```

```
Evaluating…


Word error rate:  0.34807747904018504
```

1.4.2 Question 2:

The variable `pairs_batch_test_long` contains a subset of the clean LibriSpeech test set, where the utterances are longer (longer sentences being spoken).

1. Test the performance of the model on the data stored in `pairs_batch_test_long` (use up to two decimal points). How do long utterances affect the performance of the model?

`pairs_batch_test_long`: WER=0.35. Longer utterances lower the performance of model.

2. Why do you think that is the case? longer utterances means more information in longer sentences. The words in the sentences become more distant from each other, but are still linked to each other by inforamtion overall. Base on attention mechanism, we need to consider information from further back in sentences to predict current words, which increases inaccuracy and decreases performance.

1.4.3 Question 3:

1. As a decoding strategy, we are using greedy decoding. What are the pros and cons of this?

Greedy decoding is the most straightforward apporach, which is fast and easy to understand. Each step we just select the word with highest probability. This is a greedily action sometimes output low quality words. In some cases with longer sequences. The apporach may easily get stuck on aparticular word and form dead cycles in prediction, which means repeatedly assign some special words in a cycle. It significantly affects prediction results.

2. Propose a better decoding algorithm that overcomes the downsides of greedy decoding.

Beam search decoder can be applied in training process. Compared to consider only one word in greedy decoding, now we can keep some words in a single step. Taking more words into consideration generate more hypothesis. The score of words are base on the sum log probability of these hypothesis. It shares somewhat similar ideas with N-gram in that they both include more words to increase the accuracy of the model.

1.4.4 Question 4:

In the current implementation of the `Encoder`, we are using a standard BLSTM for processing the input features.

1. What are the issues with using this type of `Encoder`?

In LSTM, when sample sequences are too long, there is a risk of losing information due to long distances between states and long time dependencies in training. The feature vectors in BLSTM contain information not only before the sequence but also after the sequence. So BLSTM requires a bidirection flow of information and computation, which increases the model training time notably.

2. How can those issues be solved?

Transformer[1] model may be applied in the task. Abandoning the LSTM network structure, Transfomers base on the encoder-decoder idea and the attention mechanism for model construction. Instead of relying on the hidden state of the past to capture dependencies on previous words, transofrmer processes a sentence as a whole, rather than word by word. So there is no risk of losing past information due to long dependencies in LSTM.

[1] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.

```
[ ]:
```