UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

DNN Speech Recognizer

REVIEW

CODE REVIEW 6

HISTORY

Meets Specifications

Answers

1. Bidirectional understanding is correct. I think you just need to look at sample code and model structure I will attach the same for you

Code

```
def final_model(input_dim, filters, kernel_size, conv_stride,conv_border_mode, units, output_dim=29, recur_layers = 2):
    """ Build a deep network for speech
    """
    # Main acoustic input
    input_data = Input(name='the_input', shape=(None, input_dim))

    # TODO: Specify the layers in your network
    #Adding Conv Layer with Batch Norm
    conv_1d = Conv1D(filters, kernel_size,
                     strides=conv_stride,
                     padding=conv_border_mode,
                     activation='relu',
                     name='conv1d_final')(input_data)
    bn_cnn = BatchNormalization(name='bn_cnn')(conv_1d)
    #Adding recurrent layer with batch norm
    simp_rnn = Bidirectional(GRU(units, activation='relu',
        return_sequences=True, implementation=2, dropout=0.1,name='bi_gru'))(bn_cnn)
    #BatchNorm
    bn_rnn = BatchNormalization(name="bn_bigru_rnn")(simp_rnn)
    for i in range(recur_layers-1):
        simp_rnn = Bidirectional(GRU(units, activation='relu',
        return_sequences=True, implementation=2, dropout=0.1,name='bi_gru' + str(i)))(bn_rnn)
        bn_rnn = BatchNormalization(name='bn_bigru_rnn' + str(i))(simp_rnn)
    time_dense = TimeDistributed(Dense(output_dim))(bn_rnn)
    # TODO: Add softmax activation layer
    y_pred = Activation('softmax', name='softmax')(time_dense)
    # Specify the model
    model = Model(inputs=input_data, outputs=y_pred)
    # TODO: Specify model.output_length
    model.output_length = lambda x: cnn_output_length(x, kernel_size, conv_border_mode, conv_stride)
    print(model.summary())
    return model
```

Model Structure

Model Structure

```python
# specify the model
model_end = final_model(input_dim=13, # change to 161 if you would like to use spect
                        filters=100,
                        kernel_size=11,
                        conv_stride=2,
                        conv_border_mode='same',
                        units=100,
                        recur_layers = 3)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
the_input (InputLayer)       (None, None, 13)          0
_________________________________________________________________
conv1d_final (Conv1D)        (None, None, 100)         14400
_________________________________________________________________
bn_cnn (BatchNormalization)  (None, None, 100)         400
_________________________________________________________________
bidirectional_1 (Bidirection (None, None, 200)         120600
_________________________________________________________________
bn_bigru_rnn (BatchNormaliza (None, None, 200)         800
_________________________________________________________________
bidirectional_2 (Bidirection (None, None, 200)         180600
_________________________________________________________________
```

I think you should have a look at these amazing blogs here

- https://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://jasdeep06.github.io/posts/Understanding-LSTM-in-Tensorflow-MNIST/

For your third question, I think a classification network should help if your outputs are limited and known. Try extracting features from model and then feed it to a classification network.

FURTHER READING IN ASR

- Guide to ASR
- ASR is devices
- Research Paper which covers different ASR models

Further Learning in NLP

I would highly recommend you to follow the current research of Transformer networks which will help you in future. To enable you, I recommend you start with this blog. It is very detailed and should be helpful. Don't get overwhelmed if you don't understand anything just keep the enthusiasm and things will fall in place.

OVERALL COMMENTS

Congratulations on finishing the project 🎉

This was a brilliant submission. You did a great job and should be proud of yourself. After reviewing this submission, I am impressed and satisfied with the effort and understanding put in to make this project a success. All the requirements have been met successfully 💯 %

I have tried to provide you a detailed review by adding :-

1. Few Suggestions which you can try and improve your model.
2. Appreciation where you did great
3. Some learning opportunities for knowledge beyond coursework

I hope you find the complete review informative :smiley: 👍

Keep doing the great work and all the best for future project.

STEP 2: Model 0: RNN

The submission trained the model for at least 20 epochs, and none of the loss values in `model_0.pickle` are undefined. The trained weights for the model specified in `simple_rnn_model` are stored in `model_0.h5`.

Simple Model works as expected.

The simple model is very naive and is not sufficient for modelling this and hence the high loss value which is perfectly fine

STEP 2: Model 1: RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `rnn_model` module containing the correct architecture.

Correct Implementation

- Good Job on implementing the correct architecture.
- You can also experiment with variants of RNN here like LSTM and also with activation functions like tanh/relu.
- Detailed variable names are provided.
- Constant naming format is followed. `model.summary()` is printed.

- Try providing inline comments as well. Good work done

The submission trained the model for at least 20 epochs, and none of the loss values in `model_1.pickle` are undefined. The trained weights for the model specified in `rnn_model` are stored in `model_1.h5`.

Performance improvement over `simple_model`

- The model performance improves by ~5x due to BatchNorm and TimeDistributed.
- Time distributed helps to apply dense layer to each timestep rather than just the final state and
- Batch normalization decreases the variance between training samples within a batch

STEP 2: Model 2: CNN + RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `cnn_rnn_model` module containing the correct architecture.

Model Overfitting :)

- You should note here that on using CNN, your training loss decreases drastically but your validation loss is not.
- A typical example of over-fitting. Had we had ran this for more epochs, the over fit would have increased. - Dropouts are a way to combat it.

- Try using MFCC as a feature, it would avoid overfitting of model.
- MFCC contains the most important features whereas spectogram contains the full breadth of features . - MFCC turns out to be better when we have a small data to train as in this case else spectogram is better for large scale fine grained training.

The main idea behind MFCC features is the same as spectrogram features: at each time window, the MFCC feature yields a feature vector that characterizes the sound within the window. Note that the MFCC feature is much lower-dimensional than the spectrogram feature, which could help an acoustic model to avoid overfitting to the training dataset.

The submission trained the model for at least 20 epochs, and none of the loss values in `model_2.pickle` are undefined. The trained weights for the model specified in `cnn_rnn_model` are stored in `model_2.h5`.

The submission trained the model for 20 epochs, and all of the loss values are defined.

STEP 2: Model 3: Deeper RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `deep_rnn_model` module containing the correct architecture.

Well done on using `recur_layers` to make your model more robust as now by a single parameter you can make you architecture as deep as you want

The submission trained the model for at least 20 epochs, and none of the loss values in `model_3.pickle` are undefined. The trained weights for the model specified in `deep_rnn_model` are stored in `model_3.h5`.

The submission trained the model for 20 epochs, and all of the loss values are defined.

STEP 2: Model 4: Bidirectional RNN + TimeDistributed Dense

The submission includes a `sample_models.py` file with a completed `bidirectional_rnn_model` module containing the correct architecture.

- Bidirectional model is implemented which helps in capturing the context on both forward and backward direction.
- Correct architecture is implemented. Good Job!!
- For how can we improve performance using Bidirectional you should read this [paper]

The submission trained the model for at least 20 epochs, and none of the loss values in `model_4.pickle` are undefined. The trained weights for the model specified in `bidirectional_rnn_model` are stored in `model_4.h5`.

The submission trained the model for 20 epochs, and all of the loss values are defined.

STEP 2: Compare the Models

The submission includes a detailed analysis of why different models might perform better than others.

- Explanation captures the essence of learning which was expected.
- Overfitting of models have been pointed out.
- You could have avoided `model_0` from graph so that the remaining graph would not have become so compressed.
- Indeed Deeper network performs better but they also have tendency to over-fit since they have huge number of params, in such scenario we should use dropout.
- All in all great work done explanation justifies the implementation.

Few tips for going above and beyond

1. You can also compare the # of parameters in each model v/s training time
2. You can also compare the # of parameters in each model v/s accuracy

STEP 2: Final Model

The submission trained the model for at least 20 epochs, and none of the loss values in `model_end.pickle` are undefined. The trained weights for the model specified in `final_model` are stored in `model_end.h5`.

The submission trained the model for 20 epochs, and all of the loss values are defined.

The submission trained the model for 20 epochs, and all of the loss values are defined.

The submission includes a `sample_models.py` file with a completed `final_model` module containing a final architecture that is not identical to any of the previous architectures.

- Final model incorporates all the understanding from above experiments and the student seems to have understood them really well. Interesting model.
- It performs pretty well. Nice usage of dropout here .

The submission includes a detailed description of how the final model architecture was designed.

- You can refer to this paper . Its simple to read and helps you grasp a deeper understanding for ASR and provides you opportunity to learn better.

DOWNLOAD PROJECT

6 CODE REVIEW COMMENTS