Part A - Information Extraction 1: Training a Named Entity Resolver [20 marks]

1. **Task 1: Create a bidirectional GRU and Multi-layer FFNN [10 marks]. In this task, you need to complete the build() method.**

```
def build(self):
  word_embeddings = Input(shape=(None,self.embedding_size,))
  word_embeddings = Dropout(self.embedding_dropout_rate)(word_embeddings)
  """
  Task 1 Create a two layer Bidirectional GRU and Multi-layer FFNN to compute the ner scores for individual tokens
  The shape of the ner_scores is [batch_size, max_sentence_length, number_of_ner_labels]
  """

  # Create a two-layer Bidirectional GRU
  # First layer
  word_output = Bidirectional(GRU(self.hidden_size,
                              return_sequences=True,
                              recurrent_dropout=self.hidden_dropout_rate))(word_embeddings)
  # Second layer
  word_output = Bidirectional(GRU(self.hidden_size,
                              return_sequences=True,
                              recurrent_dropout=self.hidden_dropout_rate))(word_output)

  # Apply dropout to the output of the GRU
  word_output = Dropout(self.hidden_dropout_rate)(word_output)

  # Create a multi-layer FFNN with two hidden layers
  # First hidden layer
  ffnn_output = Dense(self.hidden_size, activation='relu')(word_output)
  # Apply dropout to the first hidden layer
  ffnn_output = Dropout(self.hidden_dropout_rate)(ffnn_output)
  # Second hidden layer
  ffnn_output = Dense(self.hidden_size, activation='relu')(ffnn_output)
  # Apply dropout to the second hidden layer
  ffnn_output = Dropout(self.hidden_dropout_rate)(ffnn_output)

  # Output layer to compute the ner_scores
  ner_scores = Dense(len(self.ner_labels_mappings), activation='softmax')(ffnn_output)


  """
  End Task 1
  """

  self.model = Model(inputs=[word_embeddings],outputs=ner_scores)
  self.model.compile(optimizer='adam',loss="sparse_categorical_crossentropy",metrics=['accuracy'])
  self.model.summary()
```

The build method starts by creating input for word embeddings and applies dropout for regularization. Then, it constructs a two-layer Bidirectional Gated Recurrent Unit (GRU) network that processes the embeddings in both forward and backward directions, capturing contextual information from both sides. After applying dropout again, the network feeds into a Multi-layer Feed-Forward Neural Network (FFNN) with two hidden layers, each followed by dropout to further reduce overfitting. The FFNN is used to learn more advanced feature representations from the features extracted by the Bidirectional GRU. Finally, it outputs NER scores for each token using a dense layer with a softmax activation, mapping to the number of NER labels. The model is compiled with the Adam optimizer and uses sparse categorical crossentropy as the loss function.

2. **Task 2: Form the predicted named entities [10 marks].**
   a. **the code and explanation:**

```python
"""
Task 2 create the predictions of NER from the IO label
e.g.
0 I         O
1 met       O
2 John      PER
3 this      O
4 afternoon O
should give you a person NE John (x,2,2,1)
where x is the sentence id in the batch, and 2,2 are the start and end indices of the NE,
1 is the id for 'PER'
"""
predicted_labels = np.argmax(predictions, axis=-1)
for i, sent_len in enumerate(sent_lens):
      pred_entities = set()
      gold_i = {(idx, s, e, l) for idx, s, e, l in gold if idx == i}

      start = None
      current_label = None
      for j in range(sent_len):
          label = predicted_labels[i, j]
          if label != self.ner_labels_mappings['O']:
              if start is None:
                  start = j
                  current_label = label
              elif label != current_label:
                  pred_entities.add((i, start, j-1, current_label))
                  start = j
                  current_label = label
          else:
              if start is not None:
                  pred_entities.add((i, start, j-1, current_label))
                  start = None
                  current_label = None
      # Check for an entity ending at the last token
      if start is not None:
          pred_entities.add((i, start, sent_len-1, current_label))


      tp += len(pred_entities & gold_i)
      fp += len(pred_entities - gold_i)
      fn += len(gold_i - pred_entities)


"""
End Task 2
"""
```

The highest probability label indexes are obtained through np.argmax(predictions, axis=-1), representing the model's predicted labels. Then, for each sentence, the predicted labels are iterated over to construct a set of predicted named entities (pred_entities). As each word in the sentence is processed, named entities are detected and constructed. If the current label is not '0' (indicating a non-entity label), it is determined whether to start a new entity or continue the current one. Upon encountering an '0' label, the construction of the current entity ends, and it is added to the pred_entities set. For each sentence, the set of predicted entities is compared with the set of true entities to calculate the number of true positives, false positives, and false negatives. Based on these values, precision, recall, and F1 score are calculated.

b. **the F1 score achieved by model on the test set:**

   F1 score: 76.81%

c. **the output of script produces (model summary, training, and test set result)**

   model summary:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, None, 100)]       0

 bidirectional (Bidirection  (None, None, 100)         45600
 al)

 bidirectional_1 (Bidirecti  (None, None, 100)         45600
 onal)

 dropout_1 (Dropout)         (None, None, 100)         0

 dense (Dense)               (None, None, 50)          5050

 dropout_2 (Dropout)         (None, None, 50)          0

 dense_1 (Dense)             (None, None, 50)          2550

 dropout_3 (Dropout)         (None, None, 50)          0

 dense_2 (Dense)             (None, None, 5)           255

=================================================================
Total params: 99055 (386.93 KB)
Trainable params: 99055 (386.93 KB)
Non-trainable params: 0 (0.00 Byte)
```

Training result:

```
Evaluating on dev set after epoch 5/5:
F1 : 80.47%
Precision: 81.14%
Recall: 79.80%
Time used for evaluate on dev set: 0 m 2 s
```

Test set result:

```
Evaluating on test set:
F1 : 76.81%
Precision: 77.05%
Recall: 76.58%
Time used for evaluate on test set: 0 m 2 s
```

Part B - Information Extraction 2: Coreference Resolution [25 marks]

1. **Task 1: Preprocessing (See section 4.4) [10 marks].**

   Code is showed as below:

```
#  get the mentions and their cluster information.
gold_mentions, gold_mention_map, cluster_ids, num_mentions = get_mentions(clusters) # TASK 1.1 YOUR CODE HERE

# splits the mentions into two arrays, one representing the start indices,
# and the other for the end indices
raw_starts, raw_ends = zip(*gold_mentions)

# pad sentences, create glove sentence embeddings, create mention starts and ends for padded document
word_emb, starts, ends = tensorize_doc_sentences(sentences, gold_mentions) # TASK 1.2 YOUR CODE HERE

# generate (anaphor, antecedent) pairs and their labels
mention_pairs, mention_pair_labels, raw_mention_pairs = generate_pairs(num_mentions, cluster_ids, starts, ends, raw_starts, raw_ends, is_training)# TASK 1.3 YOUR CODE HERE
mention_pairs, mention_pair_labels = np.array(mention_pairs),np.array(mention_pair_labels)
```

For Task 1.1, the goal is to extract mention information and cluster IDs for each mention in a document. This has already been demonstrated in the function "get_mentions", so I directly call this function and pass the necessary arguments.

For Task 1.2, the objective is to tensorize the document sentences by padding them to a uniform length and mapping each word to its corresponding embedding. This process also requires adjusting the mention indices to reflect the padding.

For Task 1.3, the objective is to generate pairs of mentions (anaphor-antecedent pairs) and label them based on whether they belong to the same cluster (coreferent) or not.

2. **Task 2: Building the model [8 marks].**

```
def build_model():
    # 1 (a.) Initialize the model inputs
    word_embeddings = Input(shape=(None, None, EMBEDDING_SIZE), name='word_embeddings') # YOUR CODE HERE
    mention_pairs = Input(shape=(None, 4), dtype='int32', name='mention_pairs') # TASK 2.1a YOUR CODE HERE

    # squeeze the (batch_size X num_sents X num_words X embedding_size) into a
    # (num_sents X num_words X embedding_size) tensor
    word_embeddings_no_batch = Lambda(lambda x: K.squeeze(x,0))(word_embeddings)

    # 1 (b.). Apply embedding dropout to the squeezed embeddings.
    word_embeddings_dropped = Dropout(EMBEDDING_DROPOUT_RATE)(word_embeddings_no_batch) # TASK 2.1b YOUR CODE HERE

    # TASK 2.2. YOU CREATE A TWO LAYER BIDIRECTIONAL LSTM
    # word_output =
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True, recurrent_dropout=HIDDEN_DROPOUT_RATE), name='bilstm_layer_1')(word_embeddings_dropped)
    word_output = Bidirectional(LSTM(HIDDEN_SIZE, return_sequences=True, recurrent_dropout=HIDDEN_DROPOUT_RATE), name='bilstm_layer_2')(word_output)

    # flattening the lstms output and apply dropout.
    flatten_word_output = Lambda(lambda x:K.reshape(x, [-1, 2 * HIDDEN_SIZE]))(word_output)
    flatten_word_output = Dropout(HIDDEN_DROPOUT_RATE)(flatten_word_output)

    # we gather the embeddings represented by [anaphor_start, anaphor_end, antecedent_start, antecedent_end] for each pair.
    mention_pair_emb = Lambda(lambda x: K.gather(x[0], x[1]))([flatten_word_output, mention_pairs])

    # we flatten them such that each mention_pair is represented by a 400D tensor.
    ffnn_input = Reshape((-1,8*HIDDEN_SIZE))(mention_pair_emb)

    # TASK 2.3. CREATE THE MULTILAYER PERCEPTRONS THEN SQUEEZE OUT THE LAST DIMENSION USING LAMBDA
    # MLP layers
    ffnn_output = Dense(HIDDEN_SIZE, activation='relu')(ffnn_input)
    ffnn_output = Dropout(HIDDEN_DROPOUT_RATE)(ffnn_output)
    ffnn_output = Dense(HIDDEN_SIZE, activation='relu')(ffnn_output)
    ffnn_output = Dropout(HIDDEN_DROPOUT_RATE)(ffnn_output)

    # Output layer
    mention_pair_scores = Dense(1, activation='sigmoid')(ffnn_output)
    # Squeeze out the last dimension
    mention_pair_scores = Lambda(lambda x: K.squeeze(x, -1))(mention_pair_scores)


    model = Model(inputs=[word_embeddings,mention_pairs], outputs=mention_pair_scores)
    model.compile(optimizer='adam',loss='binary_crossentropy')
    print(model.summary())
    return model
```

TASK 2.1a: The word_embeddings variable accepts a tensor of shape (batch_size, num_sents, num_words, EMBEDDING_SIZE), representing the word embeddings for each word in a batch of sentences. The mention_pairs variable accepts an integer tensor of shape (batch_size, num_pairs, 4), where each quadruplet represents the start and end indices of a pair of mentions.

TASK 2.1b: A Dropout layer is applied to these word embeddings to reduce overfitting by randomly setting a fraction of the input units to zero.

TASK 2.2: Two layers of Bidirectional LSTMs are used to process the word embeddings. Bidirectional LSTMs consider both forward and backward context information, which helps in better understanding the text.

TASK 2.3: The embeddings for each mention pair are processed through two fully connected layers, each followed by a Dropout layer, using the ReLU activation function to introduce non-linearity. The final output layer is a fully connected layer that uses a sigmoid activation function to output the probability of a relationship between each mention pair. Then, a Lambda layer is used to squeeze out the last dimension, thus making the output the score of the relationship between each pair of mentions.

3. **Task 3: Coreference evaluation (Section 6.2) [4 marks].**

```
def evaluate_coref(predicted_mention_pairs, gold_clusters, evaluator):

    # turn each cluster in the list of gold cluster into a tuple (rather than a list)
    gold_clusters = [tuple(tuple(value) for value in cluster) for cluster in gold_clusters] # TASK 3.1 CODE HERE

    # mention to gold is a {mention: cluster of mentions it belongs, including the present mention} map
    mention_to_gold = {}
    # TASK 3.2 WRITE CODE HERE TO GENERATE mention_to_gold from gold_clusters
    mention_to_gold = {tuple(mention): tuple(cluster) for cluster in gold_clusters for mention in cluster}


    # get the predicted_clusters and mention_to_predict using get_predicted_clusters()
    predicted_clusters, mention_to_predicted = get_predicted_clusters(predicted_mention_pairs) # TASK 3.3 CODE HERE


    # run the evaluator using the parameters you've gotten
    evaluator.update(predicted_clusters, gold_clusters, mention_to_predicted, mention_to_gold)
```

TASK 3.1: Convert each list in the gold_clusters into tuples. Gold clusters are lists containing sets of mentions, and each mention set is also converted into a tuple. This transformation is helpful for subsequent processing because tuples are immutable and suitable as keys in dictionaries.

TASK 3.2: Use a list comprehension to create a mapping for each mention that points to the cluster it belongs to. This mapping (mention_to_gold) is a dictionary where the key is the mention (as a tuple) and the value is the cluster in which the mention resides (also in tuple form). This allows for convenient lookup of the gold standard cluster for any mention.

TASK 3.2: Use the get_predicted_clusters() function to extract the predicted clusters and the mapping from mentions to predicted clusters from the predicted mention pairs.

4. **Task 4: Some questions (Section 8)[3 marks].**

a. Would the performance decrease if we do not preprocess the text? If yes (or no), then why?

Yes, the performance will decrease if we do not preprocess the text. Preprocessing can reduce noise in the text by minimizing morphological variations. This simplification helps the model to learn the core semantics of the text more easily, without having to deal with excessive morphological changes. If preprocessing is not performed, the model will need to handle increased complexity, which requires more data to learn these additional complexities, especially in the case of limited training data. Moreover, the model might over-focus on rare variant forms, which could arise due to inconsistent use of diacritics. This could lead to decreased generalization ability of the model in practical applications.

b. Experiment with different values for max antecedent (MAX_ANT) and negative ratio (NEG_RATIO), what do you observe?

Increasing MAX_ANT can provide more historical information, which may help the model more accurately identify complex coreference relationships. However, providing more historical information may also help the model more accurately identify complex coreference relationships. Increasing NEG_RATIO may lead the model to focus more on identifying non-coreferential relationships, which can be beneficial in real-world scenarios where there are many negative samples. Decreasing NEG_RATIO can enhance the model's ability to recognize coreferential relationships, but it may perform poorly when faced with a large number of non-coreferential mentions.

c. How would you improve the accuracy?
Accuracy can be improved by the following methods: 1. Finding the optimal values for MAX_ANT (maximum number of antecedents) and NEG_RATIO (ratio of negative to positive samples), which may vary depending on the specific characteristics of the dataset. 2. Using L1 or L2 regularization to limit the size of the model parameters, helping the model generalize better to new, unseen data.

Part C - Dialogue 1: Dialogue Act Tagging [20 marks]

1. **Task 1: Implementing an utterance-based tagger, using standard text classification methods from lectures [5 marks].**

   Code is showed as below:

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Dropout, InputLayer, Bidirectional, TimeDistributed, Activation, Embedding

#Building the network
# Include 2 BLSTM layers, in order to capture both the forward and backward hidden states
model = Sequential()

# Embedding layer
model.add(Embedding(input_dim=VOCAB_SIZE, output_dim=EMBED_SIZE, input_length=128))
# Bidirectional 1
model.add(Bidirectional(LSTM(units=HIDDEN_SIZE, return_sequences=True)))
# Bidirectional 2
model.add(Bidirectional(LSTM(units=HIDDEN_SIZE, return_sequences=False)))

# Dense layer
model.add(Dense(HIDDEN_SIZE))
# Activation
model.add(Activation('softmax'))


model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

model.summary()
```

accuracy figures: 69.68981623649597

This model is a Bidirectional Long Short-Term Memory (BiLSTM) neural network. An Embedding layer maps the input indices to embedding vectors. The first LSTM layer is wrapped in a Bidirectional wrapper, allowing it to process both forward and backward information simultaneously. The `units=HIDDEN_SIZE` defines the dimension of the hidden states in the LSTM layer. `return_sequences=True` means that the hidden states for each timestep are returned, which is necessary for the subsequent BiLSTM layer. The second LSTM layer is similar to the first, but `return_sequences=False` only returns the output of the last timestep, aggregating the information into a single vector, ready for classification or other tasks. A Dense layer then transforms the LSTM output into an output vector of the same size as the hidden layer. Finally, a softmax activation function is used, allowing the final output to be interpreted as a probability distribution, suitable for multi-class classification tasks.

2. **Task 2: Minority DA tag class analysis and utterance-based tagger with re-balanced weighted cost function [5 marks].**

Code is showed as below:

```
# Re-built the model for the balanced training
model_balanced = Sequential()

# Embedding layer
model_balanced.add(Embedding(input_dim=VOCAB_SIZE, output_dim=EMBED_SIZE, input_length=128))
# Bidirectional 1
model_balanced.add(Bidirectional(LSTM(units=HIDDEN_SIZE, return_sequences=True)))
# Bidirectional 2
model_balanced.add(Bidirectional(LSTM(units=HIDDEN_SIZE, return_sequences=False)))

# Dense layer
model_balanced.add(Dense(HIDDEN_SIZE))
# Activation
model_balanced.add(Activation('softmax'))


model_balanced.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy'])

model_balanced.summary()
```

accuracy figures: 46.05380892753601

The model architecture is consistent with the one used in task1, but during training, an additional class_weight parameter is included. This parameter helps the model to account for class imbalances during the training process, making the model pay more attention to classes with fewer samples. This training strategy is aimed at enhancing the model's generalization ability on imbalanced datasets, particularly when there is a significant imbalance among the classes.

3. **Task 3: Implementing a hierarchical utterance+DA-context-based tagger [10 marks]**

Code is showed as below:

```
# BERT Embedding Layer
input_ids_in =Input(shape=(MAX_LENGTH,), dtype=tf.int32, name='input_ids')
input_masks_in =Input(shape=(MAX_LENGTH,), dtype=tf.int32, name='input_masks')
bert_embeddings = get_BERT_layer()
embedded_sent = bert_embeddings(input_ids_in, attention_mask=input_masks_in)[0]
```

```
# concatenate tensors
concatenated = Concatenate(axis=-1)([maxpool_0, maxpool_1, maxpool_2])
# flatten concatenated tensors
flatten = TimeDistributed(Flatten())(concatenated)
# dense layer (dense_1)
dense_1 = Dense(EMBED_SIZE, activation='relu')(flatten)
# dropout_1
dropout_1 = Dropout(drop)(dense_1)
dense_1.shape
```

BLSTM

This is used to create LSTM layers. The data we're working with has temporal properties which we want to model as well — hence the use of a LSTM. You should create a BiLSTM.

```
[48] # BLSTM model

     # Bidirectional 1
     BiLSTM = Bidirectional(LSTM(EMBED_SIZE, return_sequences=True))(dropout_1)
     # Bidirectional 2
     BiLSTM = Bidirectional(LSTM(units=EMBED_SIZE))(BiLSTM)

     # Dense layer (dense_2)
     dense_2 = Dense(EMBED_SIZE, activation='relu')(BiLSTM)

     flatten_2 = Flatten()(dense_2)

     # dropout_2
     dropout_2 = Dropout(drop)(dense_2)
```

Concatenate 2 last layers and create the output layer network

```
[49] # concatenate 2 final layers
     concatenated_layer = Concatenate(axis=1)([flatten_2, dropout_2])

     # output
     output = Dense(len(unique_tags), activation='softmax')(concatenated_layer)
```

Report your overall accuracy. Did context help disambiguate and better predict the minority classes ('br' and 'bf')? What are frequent errors? Show one positive example where adding context changed the prediction. layer

```
[53] # Train the model - using validation
     model2 = Model(inputs=[input_ids_in, input_masks_in], outputs=output)
     model2.summary()
     model2.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

     model2.fit(x=train_input, y=train_labels[:1000], epochs=1, batch_size=32,
                validation_data=(val_input, val_labels[:1000]), verbose=1)
```

accuracy figures: 32.60885179042816

The above code describes a complex deep learning model that integrates BERT embeddings, Convolutional Neural Networks (CNNs), and Bidirectional Long Short-Term Memory Networks (BiLSTMs). This model is particularly suited for complex natural language processing tasks that require extensive text understanding and feature extraction. Here's a detailed explanation, including the impact on minority Data Augmentation (DA) classes and how the model outputs may change:

A. **BERT Embedding Layer:** Utilizes TFDistilBertModel to obtain text embeddings from the pre-trained DistilBERT model. The BERT model captures complex contextual relationships, enriching the embedding vectors with semantic information.
B. **Convolutional and Pooling Layers:** Three different-sized convolutional kernels process the BERT output to extract features across varying scopes. Each convolutional layer is followed by batch normalization and a max pooling layer, which helps the model maintain stability across different input distributions and reduces overfitting.
C. **Bidirectional LSTM Layer:** Further processes the features post-convolution and pooling using bidirectional LSTMs to capture long-term dependencies in sequence data. This step is crucial for processing sequential data like text, as it allows the model to gather contextual information from both forward and reverse directions.
D. **Dense and Output Layers:** Transforms features through a dense layer, then outputs the probabilities for each category using a softmax activation function.

Impact on Minority DA Classes:

Minority DA classes typically refer to categories with fewer samples. Using BERT and BiLSTM can help the model better understand and leverage the linguistic features of these classes, as these technologies can capture deep semantic and sequential dependencies, potentially improving recognition rates for these classes.

Convolutional layers can extract local features from text embeddings, which may help the model better generalize and recognize rare patterns when facing categories with insufficient samples.

Changes in Model Outputs:

Without this hybrid architecture, the model might struggle to correctly classify categories with fewer samples, as traditional methods may not sufficiently capture the features of these categories.

With BERT, CNNs, and BiLSTMs, the model can provide richer and more precise feature representations, which may lead to improved prediction accuracy for minority classes. For example, if a minority class predominantly includes specific semantic or syntactic patterns, this model structure could identify these patterns through deep feature extraction.

Part D - Dialogue 2: A Conversational Dialogue System [25 marks]

1. Task 1: Implementing the encoder [5 marks].

Code:

```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units):
        super(Encoder, self).__init__()
        self.batch_sz = batch_size
        self.enc_units = enc_units
        self.embeddings = embeddings
        #  Task 1: pass the embedding into a bidirectional version of the GRU - as you can see in the call() method below,
        #  you can use just 1 GRU layer but could experiment with more

        self.Bidirectional2 = Bidirectional(GRU(self.enc_units, return_state=True))



        #  Task 1
        self.dropout = Dropout(0.2)
        self.Inp = Input(shape=(max_len_q,)) # size of questions

    def bidirectional(self, bidir, layer, inp, hidden):
        return bidir(layer(inp, initial_state = hidden))

    def call(self, x, hidden):
        x = self.embeddings(x)
#         x = self.dropout(x)
#         x = self.Bidirectional1(x)
        x = self.dropout(x)
        output, state_f,state_b = self.Bidirectional2(x)

        return output, state_f, state_b

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

Explanation:

This architecture first uses `self.embeddings` to convert the integer-encoded vocabulary into dense vector representations, followed by processing the input sequence with a bidirectional GRU in both forward and reverse directions. This allows the network to have both preceding and following contextual information at each point in the sequence, enhancing the model's ability to capture patterns that might be missed by a unidirectional approach. The `return_state=True` option in the GRU returns the final hidden states along with the output. These hidden states capture the contextual information at the last timestep of each direction, which is useful for initializing the decoder in a seq2seq model. Dropout is then applied to prevent overfitting.

This architecture is designed to maximize the understanding of the context of the input sequence, thereby effectively understanding and processing sequence data.

2. Task 2: Implementing the decoder with attention [10 marks].

Code:

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units):
        super(Decoder, self).__init__()
        self.batch_sz = batch_size
        self.embeddings = embeddings
        self.units = 2 * dec_units # because we use bidirectional encoder
        self.fc = Dense(vocab_len, activation='softmax', name='dense_layer')
        # Task 2: Create the decoder with attention - as you'll see in the call() method below, it will need two GRU layers
        self.attention = BahdanauAttention(self.units)
        self.decoder_gru_l1 = GRU(self.units,return_sequences=True,)
        self.decoder_gru_l2 = GRU(self.units, return_state=True)

        self.dropout = Dropout(0.2)


        # Task 2
    def call(self, x, hidden, enc_output):

        # enc_output shape == (batch_size, max_length, hidden_size)
        context_vector, attention_weights = self.attention(hidden, enc_output)

        # x shape after passing through embedding == (batch_size, 1, embedding_dim)
        x = self.embeddings(x)

        # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1) # concat input and context vector together

        # passing the concatenated vector to the GRU
        x = self.decoder_gru_l1(x)
        x = self.dropout(x)

        output, state = self.decoder_gru_l2(x)
        x = self.fc(output)
        return x, state, attention_weights
```

Explanation:

The decoder architecture is designed to optimize sequence generation by focusing on relevant parts of the input sequence (through attention), maintaining temporal dependencies (through GRU), and preventing overfitting (through dropout). Here are the details of the decoder architecture:

a. **Embeddings:** Converts integer-encoded vocabulary items into dense vector representations, similar to the encoder. It prepares the tokenized input for neural network processing by converting it into a format that preserves semantic relationships.

b. **Attention Mechanism:** Utilizes a Bahdanau style of attention. This component calculates the contribution of each part of the input sequence to each part of the output sequence. The attention mechanism takes the current state of the decoder and the entire output from the encoder to generate a context vector that represents the focused parts of the input sequence. This enhances the ability to effectively decode long input sequences.

c. **GRU Layers:** Uses two layers of GRU, where the first layer returns sequences, processing the concatenated context vector and embedded input. The second GRU layer processes the output from the first GRU layer but only returns the final state. This method helps capture the most relevant information at each step of the sequence, which is necessary for predicting the next output.

d. **Dropout Layer:** Applied after the first GRU layer to prevent overfitting by randomly dropping units (features) during training.
e. **Fully Connected Output Layer:** Uses a dense layer with a softmax activation function that maps the output of the second GRU layer to the size of the vocabulary. This layer produces a probability distribution over the vocabulary at each timestep, predicting the next word in the sequence.

3. Task 3: Investigating the behaviour and the properties of the encoder- decoder network [10 marks].

- **Did the models learn to track local relations between words?**

  Answer: As the number of epochs increases, the model should noticeably improve its ability to handle local context within sentences. For example, with the phrase "delicious apple pie," early training might focus on "apple pie," and as training progresses, it gradually refines its understanding of the adjective "delicious."

- **Did the models attend to the least frequent tokens in an utterance? Can you see signs of overfitting in models that hang on to the least frequent words?**

  Answer: The models might not effectively prioritize infrequent tokens, focusing more on common words to build a general language model. With more training, there is a potential risk of overfitting where the model might start focusing too much on rare words which could be less generalizable to unseen data.

- **Did the models learn to track some major syntactic relations in the utterances (e.g. subject-verb, verb-object)?**

  Answer: Early in training, syntactic understanding is limited. With more epochs, models typically begin to better understand and represent syntactic structures like subject-verb agreement and verb-object pairing.

- **Do they learn to encode some other linguistic features? Do they capture part-of-speech tags (POS tags)?**

  Answer: As the number of epochs increases, the model starts to encode other linguistic features such as part-of-speech tags, tense, and numbers implicitly through their attention mechanisms. For example, distinguishing between "running" as a verb or a noun based on the context.

- **What is the effect of more training on the length of response?**

  Answer: Typically, more training leads to better language models that can generate more contextually appropriate and length-varied responses. However, there might be diminishing returns or verbosity if not regularized. For example, early models might generate "Yes." while later models could produce "Yes, I don’t know".

- **In some instances, by the time the decoder has to generate the beginning of a response, it may already forget the most relevant early query tokens. Can**

you suggest ways to change the training pipeline to make it easier for the model to remember the beginning of the query when it starts to generate the response?

Answer: Enhancing focus on the early parts of a query can be achieved through methods such as integrating memory-augmented networks and introducing or increasing cross-attention layers in the decoder. However, the specific implementation depends on technical feasibility and the specific objectives of the dialogue system.

Part E - Prompt Engineering [10 marks]

1. **Task 1: Natural Language Generation [2 marks]**

Code:

```
prompt_task1 ="Write a recipe for spaghetti al pomodoro" # YOUR PROMPT HERE
prompt_template_task1 = f'''

USER: {prompt_task1}

ASSISTANT:
'''
```

Prompt Result:

```
USER: Write a recipe for spaghetti al pomodoro

ASSISTANT:

Spaghetti al Pomodoro is a classic Italian dish that originated in the Tuscany region. It's made with fresh tomatoes, garlic, olive oil, and of course, spaghetti! Here's my recipe for this delicious pasta dish:

Ingredients:

* 12 oz (340g) spaghetti
* 2 large ripe tomatoes, peeled and chopped
* 3 cloves garlic, minced
* 6 tbsp (84g) extra-virgin olive oil
* Salt and freshly ground black pepper to taste
* Fresh basil leaves for garnish (optional)

Instructions:

1. Bring a large pot of salted water to a boil. Cook the spaghetti according to package instructions until al dente. Reserve 1 cup of pasta cooking water before draining the spaghetti.
2. In a blender or food processor, combine the tomatoes, garlic, and olive oil. Blend until smooth, adding some reserved pasta cooking water if needed to achieve
```

Explanation:

In the zero-shot prompting scenario, LLaMA 2 needs to understand what “spaghetti al pomodoro” is and generate a relevant recipe. This requires the model to have sufficient knowledge of food and an understanding of recipe structures. The response needs to have the format and style of a recipe, including a list of ingredients and detailed cooking instructions. Although this task was not used directly in training Llama2, the model’s zero-shot capability allows it to generate the required content without explicit examples.

2. **Task 2: Summarization [2 marks]**

Code:

```
prompt_task2 = f'''Summarize the Wikipedia article you find in the following lines, beginning with ARTICLE-BEGIN and ending with ARTICLE-END
ARTICLE-BEGIN

{task2_article}

ARTICLE-END
'''# YOUR PROMPT HERE

prompt_template_task2 = f'''

USER: {prompt_task2}

ASSISTANT:
'''
```

Prompt Result:

ASSISTANT:
The article discusses LLaMA, a family of autoregressive large language models (LLMs) released by Meta AI in February 2023. The models are available under a noncommercial license and have been shown to perform well on various natural language processing benchmarks. The article also highlights the differences between LlaMA and other large language models, such as GPT-3, including the use of SwiGLU activation function instead of GeLU, rotary positional embeddings instead of absolute positional embedding, and root-mean-squared layer-normalization instead of standard layer-normalization. Additionally, the article discusses the training datasets used for LlaMA, which include a large variety of publicly available data sources such as web pages, open source repositories, Wikipedia, and scientific papers. Finally, the article mentions the release of LlaMA-2 in July 2023 with improved performance and additional fine-tuned models for dialog.

Explanation:

In this task, LLaMA 2 needs to parse the instructions within the prompt. The model will process and analyze the text between “ARTICLE-BEGIN” and “ARTICLE-END”. This requires advanced text comprehension skills to capture the main ideas and events of the article. Based on the content of the article, the model needs to generate a concise summary, which may include identifying characters, locations, events, and any significant dates or data. The summary should be succinct and retain the core information of the article.

3. **Task 3: Machine Translation [2 marks]**

Code:

```
prompt_task3 = f'''Translate to Italian the Wikipedia article you find in the following lines, beginning with ARTICLE-BEGIN and ending with ARTICLE-END
ARTICLE-BEGIN

{task2_article}

ARTICLE-END
'''# YOUR PROMPT HERE
prompt_template_task3 = f'''

USER: {prompt_task3}

ASSISTANT:
'''
```

Prompt Result:

ASSISTANT:
Hello! I'd be happy to help you translate the article from English to Italian. Please keep in mind that the translation may not be perfect and may require some adjustments to accurately convey the meaning of the text. Here is the translation of the article beginning with ARTICLE-BEGIN and ending with ARTICLE-END:

ARTICLE-BEGIN

LLaMA (Large Language Model Meta AI) è una famiglia di modelli di linguaggio liberi autoregolari, rilasciati da Meta AI a partire da febbraio 2023.

Per la prima versione di LLaMA, quattro taglie di parametri sono state utilizzate: 7, 13, 33 e 65 miliardi.
I sviluppatori di LLaMA hanno riferito che il modello con 13 parameteri ha eseguito migliori prestazioni su molti benchmark NLP rispetto al molto più grande GPT-3 (con 175 parametri) e che il modello più grande era competitivo con i modelli di stato dell'arte come PaLM e Chinchilla.[1]

Explanation:

In this task, while translating the entire article, LLaMA 2 needs to maintain the accuracy of the original structure and content, including correctly translating proper nouns, technical terms, and culture-specific concepts. Based on the results, the model demonstrates a strong capability for language generation, producing coherent sentences that conform to the conventions of the target language.

4. Task 4: Named Entity Recognition [2 marks]

Code:

```
prompt_task4 = f'''Identify the named entities in the sentence in the following lines (between with SENTENCE-BEGIN and SENTENCE-END), and tag them as
ORG if organization (for instance, Apple, IBM)
DATE if date (for instance, 8/4/2024)
NUM if number (for instance, 65)
MODEL if an AI model (for instance, ChatGPT)
Output the result in JSON format.

SENTENCE-BEGIN
{task4_sentence}
SENTENCE-END
'''# YOUR PROMPT HERE
prompt_template_task4 = f'''

USER: {prompt_task4}

ASSISTANT:
'''
```

Prompt Result:

```
ASSISTANT:
{
"entities": [
{"label": "ORG", "value": "Meta"},
{"label": "DATE", "value": "July 18, 2023"},
{"label": "NUM", "value": "7, 13, and 70 billion parameters"}
]
}
```

Explanation:

In this task, LLaMA 2 needs to be able to recognize and differentiate various types of named entities. The provided examples are crucial for ensuring that the model correctly understands the task requirements and output format. These examples serve the role of training samples for the model. Although they are few in number, they are sufficient to guide the model in executing the task correctly.

5. Task 5: Dialogue Act Tagging [2 marks]

Code:

```
prompt_task5 =f'''Identify the dialogue acts in the conversation below between A and B (starting from CONVERSATION-BEGIN and ending with CONVERSATION-END), using the following tags:
QW if the utterance is a wh-question (for instance, where are you going?)
SD if the utterance is a statement (for instance, I am going to London)
ACK if the utterance is an acknowledgment (for instance, OK)
Output the result in JSON format.

CONVERSATION-BEGIN

{task5_conversation}

CONVERSATION-END
'''# YOUR PROMPT HERE
prompt_template_task5 = f'''

USER: {prompt_task5}

ASSISTANT:
'''
```

Prompt Result:

```
ASSISTANT:

{
"acts": [
{
"type": "QW",
"utterance": "And, what day in May did you want to travel?"
},
{
"type": "SD",
"utterance": "OK uh I need to be there for a meeting that's from the 12th to the 15th."
},
{
"type": "ACK",
"utterance": "And you're flying into what city?"
},
{
"type": "SD",
"utterance": "Seattle."
},
{
"type": "QW",
"utterance": "And what time would you like to leave Pittsburgh?"
},
{
"type": "ACK",
"utterance": "Uh hmm I don't think there's many options for non-stop."
},
{
"type": "SD",
"utterance": "Right. There's three non-stops today."
}
]
}
```

Explanation:

In this task, LLaMA 2 needs to parse the content of the conversation, identify the intent and function of each utterance, and output the results in JSON format. This requires LLaMA 2 to accurately categorize each utterance in the conversation based on the provided instructions and examples, and to format the output as specified. The final results indeed meet the requirements.