Transformers for NLP Tasks

presented by

Rahul Reddy Talatala (UBIT: rtalatal)

Revanth Siva Sai Ram Balineni (UBIT: rbalinen)

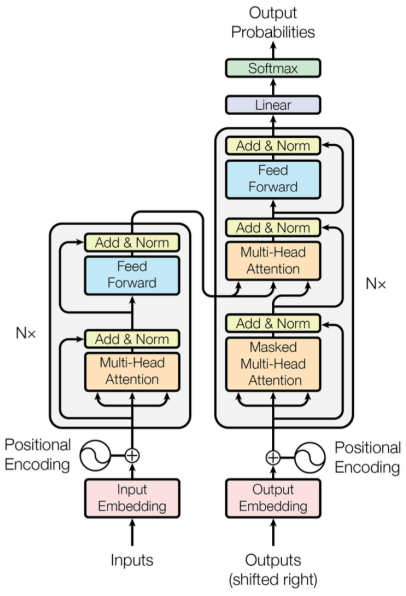
Transformers

Why Transformers came into the picture?

- RNN can not be parallelized as each step in relies on the output of the previous step.
- RNNs tend to forget the beginning of the sentence by time it reaches the end of the sentence.
- RNN and LSTM are sequential models i.e. they process the data sequentially, it hinders the parallelization process and leads to slower training times.
- RNNs struggle to capture long-range dependencies due to the vanishing gradient problem, where gradients become extremely small as they are propagated backward through the network layers. This hinders the learning of the network due to the diminishing impact of the earlier hidden states.
- In the context of RNNs, the vanishing gradient problem arises when the error signal used to train the network decreases exponentially as it propagates backward through time, causing layers closer to the input to receive minimal training.
- LSTMs prevent the vanishing gradient issue by enabling better control over gradient values at each time step through suitable parameter updates of forget gates. But, LSTMs are computationally expensive, memory intensive and prone to overfitting.

About Transformers

- Transformers only rely on attention mechanism and doesn't have any recurrence, so we can train them in parallel.
- Transformers are generally made up of encoder and decoders. The left part is the encoder and the right part is the decoder (refer to the below figure).



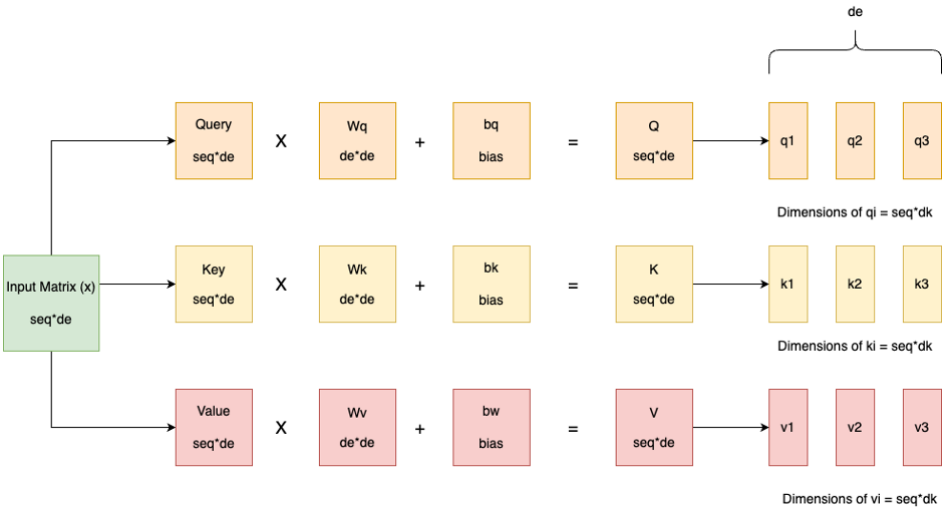
Transformer's Architecture

- Attention is the ability of the model to pay attention to the important part of the input
- Transformers basically have a 6 encoders and 6 decoders
- Each encoder has 1 self attention layer and 1 Feed Forward Neural Network Layer

- Each decoder has 2 self attention layers and 1 Feed Forward Neural Network Layer
- The parallelization happens on how we feed the input to the transformer. We feed all the words of the sentence to the encoder.
- All the words in the multi-head attention layer is compared to all the other words in the same layer.
- Later each word is passed separately to each Feed Forward Neural Network. They don't have any internal communication. The architecture of these internal Feed Forward Neural Network are same in the same Encoder layer, **But the architecture of FNN varies from one encoder to another encoder.**
- As transformers do not have any recurrence they do not know which words comes first and which comes later. So they use **positional encoding,** we inject information with each word about where this word occurs in the sentence.
- We have a linear layer and a softmax layer at the end of each decoder. So after these layers, it produces a vector that is the size of the vocabulary and each of these cells in this vector tells us **how likely this word in this cell is going to be the next word in the sequence.**
- Transformers are different from other architectures as they have add and normalization layers. This layer normalizes the input that comes from the sub layer and it is called as layer normalization, which is an improved version of batch normalization.
- Transformers have skip states (arrows in the above figure), some of the information that does not go to the attention layers or the FNN layers, it directly goes to the normalization layers. This helps the model to not forget things and helps the model to forward the information that is important to further in the network
- Inside these normalization layers, we add the information from that went to the sub-layer and skip the sub-layer.

Architecture Breakdown

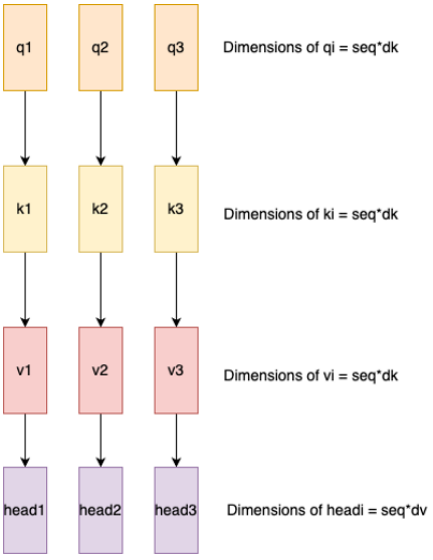
1. **Linear Transformations**: The input sequence of words is first converted into numerical representations called embeddings, where each word is represented as a vector of a specified dimension. To incorporate positional information, since transformers lack recurrence, positional encodings are added to the word embeddings. The resulting input matrix is then linearly transformed into **Query**, **Key**, and **Value** matrices by multiplying it with learnable weight matrices and adding bias terms. These linear transformations project the input into different representations that can capture different aspects of the input data.



Linear Transformations to generate Query, Key, Value

2. **Scaled Dot Product Attention**: The scaled dot-product attention mechanism is at the core of the transformer architecture. It computes attention scores by taking the dot product of the Query and Key matrices, scaled by the square root of the key dimension for normalization purposes. This scaling helps prevent the dot products from becoming too large or too small, which can lead to gradient issues during training. The attention scores are then used to compute a weighted sum of the Value vectors, effectively allowing the model to attend to different parts of the input sequence when generating the output representation.

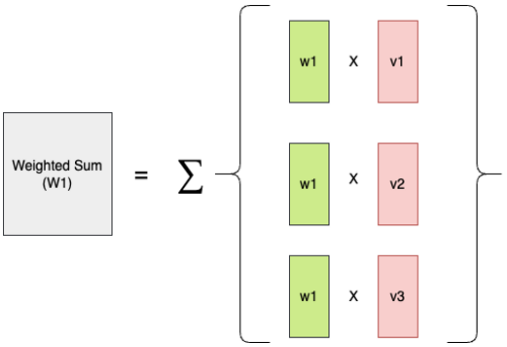
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



Multi-head attention computation

3. **Weighted Sum:** The weighted sum is the final step in the attention mechanism. After computing the attention scores (weights) using the scaled dot-product operation between the Query and Key matrices, these weights are multiplied with their corresponding Value vectors. The weighted sum of these Value vectors is then calculated, where each Value vector is weighted by its corresponding attention score. This weighted sum represents the attended output, capturing the most relevant

information from the input sequence based on the learned attention scores. The weighted sum is the output of the attention layer and serves as input to subsequent layers in the transformer architecture.



4. **Optional Output Transformation:** After computing the weighted sum of the value vectors through the attention mechanism, an optional output transformation can be applied to the resulting matrix. This transformation involves multiplying the weighted sum matrix by an additional learnable weight matrix (WO) and adding a bias term (bO). Mathematically, this can be represented as: `z = Weighted Sum * WO + bO`

Task 1 - Text Summarization by BART

DialogSUM Dataset

For the text summarization task, the DialogSUM dataset is used. It is a large-scale dialogue summarization dataset that contains 13,460 dialogues. The dialogues cover a wide range of daily-life topics, including schooling, work, medication, shopping, leisure, and travel.

This is how a sample dialogue and summary looks like:

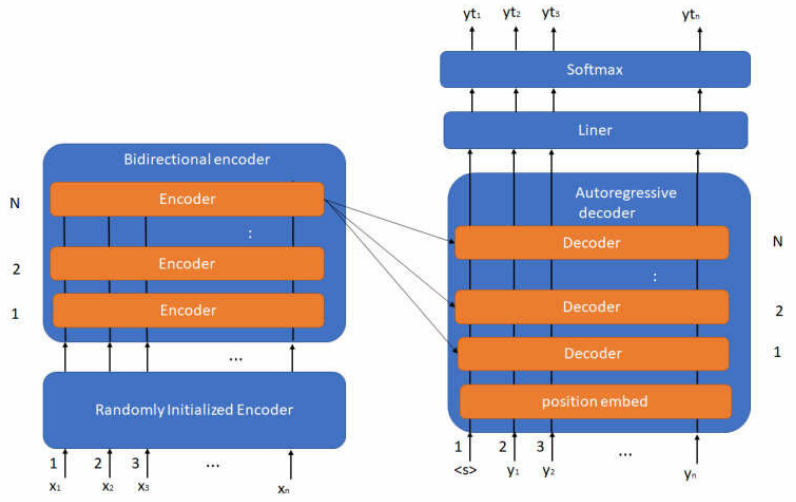
```
Dialogue:
#Person1#: Hi, Mr. Smith. I'm Doctor Hawkins. Why are you here today?
#Person2#: I found it would be a good idea to get a check-up.
#Person1#: Yes, well, you haven't had one for 5 years. You should have one every y
#Person2#: I know. I figure as long as there is nothing wrong, why go see the doct
#Person1#: Well, the best way to avoid serious illnesses is to find out about them
#Person2#: Ok.
#Person1#: Let me see here. Your eyes and ears look fine. Take a deep breath, plea
#Person2#: Yes.
#Person1#: Smoking is the leading cause of lung cancer and heart disease, you know
#Person2#: I've tried hundreds of times, but I just can't seem to kick the habit.
#Person1#: Well, we have classes and some medications that might help. I'll give y
#Person2#: Ok, thanks doctor.

Summary:
Mr. Smith's getting a check-up, and Doctor Hawkins advises him to have one every y
```

```
Topic:
get a check-up
```

Now let's implement the text summarization task using BART model

About BART model

BART (Bidirectional and Auto-Regressive Transformers) is an encoder-decoder model that uses a bidirectional encoder and an auto-regressive decoder, the bidirectional encoder helps capture the full context, while the auto-regressive decoder generates the output in a natural, flowing manner. This unique architecture enables BART to be a powerful tool for text generation, excelling at tasks such as summarization, translation, and open-ended writing.



BART Architecture

BART has an encoder that can grasp the context of input sequences. This trait is vital for summarizing content. The decoder produces summaries token-by-token, considering previous tokens and the encoder's context. BART excels at summarizing texts because it adeptly handles long inputs and generates cohesive, concise summaries. Its ability to capture long-range dependencies through attention mechanisms makes it well-suited for this task.

Step 1: Tokenizing Dataset

First, the data needs tokenization before inputting into the model. The `tokenizeDataset` function tokenizes the DialogSUM dataset. It tokenizes dialogues using the BARTTokenizer. A 750 token limit is set - exceeding that the dialogues get truncated. Next, it tokenizes summaries via `as_target_tokenizer`, ensuring tokenizer generates token IDs for target sequences (summaries). The tokenized summary sequences become labels for dialogue input sequences, creating a dictionary for dataset creation.

```
#this function is used to tokenize the dataset
def tokenizeDataset(dataset, tokenizer, diagLength=750, summaryLength=150):
  #tokenising the dialogues in the dataset and fetching the tensors
  #specifying the max length of the dialogues to be 750 and if a dialogue is longe
  dialogues = tokenizer(dataset["dialogue"], max_length=diagLength, truncation=Tru
  #setting the tokenizer to genreate token IDs for the target sequences
```

```
  with tokenizer.as_target_tokenizer():
    #tokenising the dialogues in the dataset and fetching the tensors
    #specifying the max length of the dialogues to be 150 and if a dialogue is lon
    summaries = tokenizer(dataset["summary"], max_length=summaryLength, truncation
  #assigning the tokenised summary sequences as labels for the dialogue input sequ
  dialogues["labels"] = summaries["input_ids"]
  return Dataset.from_dict(dialogues)

 #loading the BART tokenizer
tokenizer = BartTokenizer.from_pretrained('facebook/bart-large-xsum')
```

Step 2: Creating pipeline for summarization

The following code snippet is used for creating a pipeline for the summarization task using the pre-trained BART model `facebook/bart-large-xsum` with the `pipeline` function from the Hugging Face Transformers library.

```
#creating a pipeline for summarisation task using the BART model
dialogSummariser = pipeline('summarization', model = 'facebook/bart-large-xsum')
```

Step 3: Importing the pre-trained BART model

The following code snippet imports a pretrained BART model called `facebook/bart-large-xsum`. This was done with BartForConditionalGeneration from Hugging Face Transformers. The model's complex architecture enabled it to generate concise summaries from lengthy text inputs.

```
#loading the pre-trained BART model
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large-xsum')
```

Step 4: Collating the data

The following code snippet initializes a `DataCollatorForSeq2Seq` object, which is used to process batches of data for the model. This collator ensures that the input dialogues and output summaries are properly formatted and padded to the maximum sequence lengths.

```
#initialising an collater to process batches of data for the model
dataCollator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=model)
```

Step 5: Loading the metrics

The following code snippet contains the `computeMetrics` function which is used to compute various evaluation metrics for the model's predictions and the actual truth labels. This function first unpacks the predictions and labels, decodes them using the tokenizer, and tokenizes the sentences for computing `ROUGE` and `BLEU` scores. Additionally, the function also computes the `mean generated summary length` by counting the non-padding tokens in the predictions.

```
#loading metrics for evaluating the model
rougeScore = load_metric('rouge')
bleuScore = load_metric('bleu')

#this function is used for computing various evaluation metrics
def computeMetrics(data):
  #unpacking the predictions and labels
  predictions, labels = data
  #decoding predictions
  decodedPreds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
  #decoding true labels and eliminating masked tokens (label = -100)
  decodedLabels = tokenizer.batch_decode(np.where(labels != -100, labels, tokenize
  #tokenizing sentences for computing rouge score
  decodedPreds = ["\n".join(sent_tokenize(pred.strip())) for pred in decodedPreds]
  decodedLabels = ["\n".join(sent_tokenize(label.strip())) for label in decodedLab
  #tokenizing sentences into a list for computing bleu score
  bleuDecodedPreds = [word_tokenize(pred.strip()) for pred in decodedPreds]
  bleuDecodedLabels = [[word_tokenize(label.strip())] for label in decodedLabels]
  #computing ROUGE scores
  rouge = rougeScore.compute(predictions=decodedPreds, references=decodedLabels, u
  #computing BLEU scores
  bleu = bleuScore.compute(predictions=bleuDecodedPreds, references=bleuDecodedLab
  #combining all metric scores into a single dictionary
  result = {
      #rouge scores are returned as a dictionary, so extracting ROUGE F1 scores
      **{key: value.mid.fmeasure * 100 for key, value in rouge.items()},
      **bleu
  }
  #computing mean generated summary length
  predLengths = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in pred
  result["gen_len"] = np.mean(predLengths)
  return result
```

Step 6: Defining Training Arguments and Trainer

The following code snippet then configures the training by defining a `Seq2SeqTrainingArguments` object. This object specifies various training parameters, such as the output directory, evaluation strategy, save strategy, learning rate, batch size, gradient accumulation steps, weight decay, number of epochs, mixed precision, and more.

```
#configuring the training process
trainingArgs = Seq2SeqTrainingArguments(
    #directory to save the model's checkpoints
    output_dir = "/content/drive/MyDrive/DL Code Demo/bart_output",
    #evaluting the model after end of each epoch
    evaluation_strategy = "epoch",
    #saves the model checkpoint at the end of each epoch
    save_strategy = "epoch",
    #loading the best model at the end of each epoch
```

```
    load_best_model_at_end = True,
    #metric for choosing the best model
    metric_for_best_model = "eval_loss",
    seed = 36,
    learning_rate=1e-5,
    #setting the batch size for training and evaluation
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    #accumulates gradient for 2 steps before updating the weights
    gradient_accumulation_steps=2,
    #applying regularisation
    weight_decay=0.01,
    #saving only 1 checkpoint during training
    save_total_limit=1,
    #no of epochs to train
    num_train_epochs=2,
    predict_with_generate=True,
    #enabling mixed precision for faster computation
    fp16=True,
    report_to="none"
)
```

Next, the following code snippet defines a `Seq2SeqTrainer` object, which is responsible for training the model based on the specified training arguments. The trainer is initialized with the model, training arguments, tokenized datasets, tokenizer, data collator, and the `computeMetrics` function for computing evaluation metrics.

```
#defining the model's trainer
trainer = Seq2SeqTrainer(
    model=model,
    args=trainingArgs,
    train_dataset=trainTokenizedDataset,
    eval_dataset=testTokenizedDataset,
    tokenizer=tokenizer,
    data_collator=dataCollator,
    compute_metrics=computeMetrics,
)
```

Step 7: Evaluating the Model

After training the model, the following code snippet evaluates the model's performance on the tokenized validation and test datasets using the `evaluate` method of the `Seq2SeqTrainer`. This method computes the evaluation metrics specified in the `computeMetrics` function.

Evaluating the model on the validation dataset

```
Loss: 0.2515592873096466
Rouge1: 49.69853771137852
Rouge2: 25.16883886800042
RougeL: 41.51005648279664
```

```
RougeLsum: 44.80822164023037
BLEU Score: 0.2333263389348458
```

Evaluating the model on the test dataset

```
Loss: 0.2703404724597931
Rouge1: 45.18382459222041
Rouge2: 19.777718288950535
RougeL: 36.6363668266085
RougeLsum: 39.86764225453036
BLEU Score: 0.17471813311109283
```

Based on the evaluation metrics for the validation and test datasets, we can draw the following insights:

1. **Loss**: The loss on the test dataset `0.2703` is slightly higher than the loss on the validation dataset `0.2516`, indicating that the model's performance is slightly worse on the unseen test data compared to the validation data. This difference in loss values suggests that the model has generalized reasonably well to the test set.
2. **ROUGE Scores**:
   - The ROUGE-1 score measures the unigram overlap, this score is higher on the validation set `49.70` compared to the test set `45.18`
   - The ROUGE-2 score measures bigram overlap is also higher on the validation set `25.17` compared to the test set `19.78`, further confirming better performance on the validation data.
   - The ROUGE-L score measures the longest common subsequence and ROUGE-L-SUM score ROUGE-L with a recall score for summary-level evaluation follow a similar trend, with higher scores on the validation set `41.51` and `44.81`, respectively compared to the test set `36.64` and `39.87`, respectively.
3. **BLEU Score**: It measures the n-gram overlap between the generated summaries and reference summaries, is higher on the validation set `0.2333` compared to the test set `0.1747`.

The validation set is slightly similarly to the training data when compared to the test set, thus leading to better generalization on the validation set.

Model generated summary on Seen Data

```
Original Dialogue:
#Person1#: Can I help you, ma'am.
#Person2#: Yes, will you keep our bags until 6 p. m. ?
#Person1#: Are you all our guests?
#Person2#: Yes, we checked out just now.
#Person1#: Please fill in this form.
#Person2#: Is this all right?
#Person1#: So you are leaving this evening. We keep your bags until 6 p. m. Here's
#Person2#: Thanks a lot.
```

```
Orignal Summary:
 #Person2# asks for #Person1#'s help to keep the bags until 6 p.m.

Generated Summary:
'A group of people have been asked to leave their bags at a hotel in Paris.'
```

Model generated summary on Unseen Data

```
Original Dialogue:
#Person1#: Miss Chen, I'll go to America to spend my holiday. Would you please boo
#Person2#: All right, Mr. Kinite. A telex message from London has just come in. Sh
#Person1#: Yes, please give it to me now. By the way, any orders today?
#Person2#: Yes. I've forwarded them to the Sales Department. Here's the telex mess
#Person1#: Thank you.
#Person2#: Shall I hold your business mail while you are gone or handle the letter
#Person1#: Handle the letters yourself and keep them on files. But forward private
#Person2#: I see.
#Person1#: That will be all for now. I'll call on you if I need anything else.

Generated Summary:
'A group of people are sitting in a conference room at the headquarters of a Chine
```

Activity Task

Use the same DialogSUM dataset to fine-tune the PEGASUS Model for text summarization. Refer to the documentation to implement the task.

Task 2 - Named Entity Recognition using BiLSTM

NER Dataset

This dataset, derived from the GMB corpus, is an annotated corpus for Named Entity Recognition (NER), facilitating entity classification through Natural Language Processing. With over a million tagged entities covering various geographical, organizational, and personal categories, it offers a rich resource for training classifiers. Its recent enhancements and diverse features make it invaluable for tasks like extracting custom-named entities from texts, thereby aiding in solving real-world business challenges, such as parsing Electronic Medical Records.

We approach this dataset using a BiLSTM Model.

About the BiLSTM Model

The Bidirectional Long Short-Term Memory (BiLSTM) model is a recurrent neural network (RNN) architecture that captures long-range dependencies in sequential data. Unlike traditional LSTM models, which process input sequences in only one direction, BiLSTMs simultaneously process input sequences in both forward and backward directions. This bidirectional approach enables the model to effectively capture contextual information from both past and future inputs, enhancing its ability to understand the context of each word or token in a sequence.

In the case of Named Entity Recognition (NER), the goal is to identify and classify entities by identifying names of people, organizations, and locations within a given text. In this scenario, a BiLSTM model would analyze the input sentence by simultaneously processing it from left to right and from right to left. This allows the model to capture the preceding context and the succeeding context for each word, enabling it to make more accurate predictions about whether a particular word is part of a named entity. By leveraging the bidirectional nature of the BiLSTM architecture, the model can effectively recognize entities even in complex and context-dependent sentences.

Step 1: Preprocessing

The steps involved in preprocessing are:

1. **Loading Word Embedding Model**: The code first loads the Word2Vec model from the Google News dataset using the gensim library. This model represents words as dense vectors and is pre-trained on a large corpus of text.
2. **Reading and Preprocessing the Dataset**: The code reads a dataset stored in a CSV file using pandas. It handles missing values in the 'Sentence #' column by forward filling them with the previous non-null value. Then, it renames columns for ease of use and converts all tokens to lowercase to ensure case matching.
3. **Grouping Sentences and Tokenizing**: The DataFrame is grouped by the 'Sent' column, and each sentence is tokenized along with its respective part-of-speech tag. These tokenized sentences are stored as lists of tuples.
4. **Defining Word and Tag Indexes**: Unique words and tags are extracted from the dataset, and each is assigned a unique index. Special tokens for unknown words ('UNK') and padding ('PAD') are also included.
5. **Creating Word Embedding Matrix**: An embedding matrix is initialized, and each word in the vocabulary is looked up in the Word2Vec model. If a word is found, its corresponding embedding vector is added to the matrix. If not, a zero vector is used.
6. **Padding Sequences and Splitting Data**: The tokenized sequences are padded to a maximum length to ensure uniform input size for neural network training. The dataset is split into training and testing sets using the train_test_split function.
7. **One-Hot Encoding Tags**: The tags are converted into one-hot encoded vectors to match the model's output format.

These preprocessing steps prepare the dataset for training a neural network, ensuring compatibility with the chosen architecture and optimizing its performance during training.

```
words = list(set(df['tkn'].values))
tags = list(set(df["pos_ind"].values))

word2idx = {w: i + 2 for i, w in enumerate(words)}
word2idx.update({"UNK": 1, "PAD": 0})
```

```
tag2idx = {t: i+1 for i, t in enumerate(tags)}
tag2idx["PAD"] = 0

hits, misses = 0, 0
embedding_dim = 300

import numpy as np

embedding_matrix = np.zeros((len(word2idx), embedding_dim))
#print(len(embedding_matrix))
#print(word2idx)
for word, i in zip(word2idx.keys(), word2idx.values()):
    try:
        embedding_vector = model[word]
        embedding_matrix[i] = embedding_vector
        hits += 1
    except Exception:
        misses += 1
        pass
```

Step 2: Dataset Preparation

The Dataset preparation is denoted in the code snippet attached below. Here's a breakdown of the process:

1. **Tokenization and Indexing**: The first step involves converting the tokenized sentences (`sentence_all`) into sequences of indices based on the word-to-index (`word2idx`) and tag-to-index (`tag2idx`) mappings. This is achieved using list comprehensions, where each token (`w[0]`) is looked up in the word-to-index dictionary (`word2idx`). If the token is not found, it defaults to index 1, representing the "UNK" token.
2. **Padding Sequences**: Since neural networks require inputs of uniform length, sequences are padded or truncated to a fixed length (`MAX_LEN`). The `pad_sequences` function from Keras preprocessing is used for this purpose. Both the input sequences (`x`) and output sequences (`y`) are padded with zeros at the end (post-padding) to match the maximum length.
3. **One-Hot Encoding**: The output sequences (`y`) are further processed to convert them into one-hot encoded vectors. Each index representing a tag is converted into a binary vector where the index corresponding to the tag is set to 1, and all other indices are set to 0. This is achieved using the `to_categorical` function from Keras utilities.
4. **Splitting Data**: Finally, the dataset is split into training and testing sets using the `train_test_split` function from scikit-learn. This function randomly shuffles and splits the input data (`x`) and output data (`y`) into training and testing subsets, with the specified test size (0.1 in this case).

This process ensures that input and output data are appropriately formatted and preprocessed for training a neural network model, enabling efficient learning and evaluation.

```
X = [[word2idx.get(w[0], 1) for w in s] for s in sentence_all]
X = pad_sequences(maxlen=MAX_LEN, sequences=X, padding="post", value=0)
```

```
y = [[tag2idx[w[1]] for w in s] for s in sentence_all]
y = pad_sequences(maxlen=MAX_LEN, sequences=y, padding="post", value=0)
y = [to_categorical(i, num_classes=len(tags)+1) for i in y]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1)
```

Step 3: Building a BiLSTM model

This code builds a Bidirectional Long Short-Term Memory (BiLSTM) model for Named Entity Recognition (NER). Let's break down the code:

1. **Input Layer**: The code starts by defining an input layer for the model, specifying the shape of the input data. In this case, the input shape is set to `(max_len)`, where `max_len` represents the maximum length of input sequences.
2. **Embedding Layer**: The input sequences are passed through an embedding layer. This layer converts input sequences of integer indices into dense vectors of fixed size (`embedding_dim`). The embedding layer is initialized with pre-trained word embeddings (`embedding_matrix`) to capture semantic information from the input tokens.
3. **Bidirectional LSTM Layers**: The embedded sequences are processed by multiple layers of Bidirectional Long Short-Term Memory (BiLSTM) units. Each BiLSTM layer processes the input sequences in both forward and backward directions, allowing the model to capture contextual information from past and future tokens. The code has three stacked BiLSTM layers, each returning sequences (`return_sequences=True`) to pass information to the subsequent layers.
4. **Dense Layer**: After the BiLSTM layers, a TimeDistributed Dense layer is applied to add non-linearity to the output sequences. This layer applies a dense neural network layer to each timestep of the input sequences independently, allowing the model to learn complex patterns in the sequence data.
5. **CRF Layer**: The output of the Dense layer is passed through a Conditional Random Field (CRF) layer. The CRF layer models the dependencies between output labels in the sequence, considering the entire sequence when assigning labels to individual tokens. This helps capture label dependencies and improves the overall sequence labeling performance.
6. **Model Compilation**: The model is compiled with an optimizer (Adam) and a loss function (SigmoidFocalCrossEntropy). Adam optimizer is used with a learning rate of 0.001, and the SigmoidFocalCrossEntropy loss function is employed, which is suitable for multi-class classification tasks.
7. **Model Building**: The model is constructed using the `Model` class from Keras, with the input and CRF layers as inputs and outputs, respectively.
8. **Return Model**: The function returns the compiled model instance.

This code defines a BiLSTM-CRF model architecture suitable for sequence labeling tasks and compiles it for training with appropriate optimization and loss settings.

```
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Embedding, Dense, TimeDistributed
```

```
from tensorflow_addons.layers import CRF
from tensorflow_addons.losses import SigmoidFocalCrossEntropy
from tensorflow.keras.optimizers import Adam

def build_model(max_len=75, input_dim=len(word2idx), embedding_dim=300, num_tags=1
    # Define input layer
    input_layer = Input(shape=(max_len,))

    # Embedding layer
    embedding_layer = Embedding(
        input_dim=input_dim,
        output_dim=embedding_dim,
        weights=[embedding_matrix],
        input_length=max_len,
        mask_zero=True,
        trainable=True,
        name='embedding_layer'
    )(input_layer)

    # Bidirectional LSTM layers
    lstm_1 = Bidirectional(LSTM(units=50, return_sequences=True))(embedding_layer)
    lstm_2 = Bidirectional(LSTM(units=100, return_sequences=True))(lstm_1)
    lstm_3 = Bidirectional(LSTM(units=50, return_sequences=True))(lstm_2)

    # Dense layer
    dense_out = TimeDistributed(Dense(25, activation="relu"))(lstm_3)

    # CRF layer
    crf_layer = CRF(num_tags, name='crf')(dense_out)

    # Build model
    model = Model(input_layer, crf_layer)

    # Compile model
    model.compile(
        optimizer=Adam(learning_rate=0.001),
        loss=SigmoidFocalCrossEntropy()
    )

    return model

model = build_model()
```

Step 4: Training Process:

Below is a breakdown of the training process:

1. **Model Definition**: The model architecture is defined using the `build_model()` function, which creates a neural network model for Named Entity Recognition (NER) based on the specified parameters.
2. **Callbacks Setup**: Two callbacks are set up:

- `ModelCheckpoint`: This callback monitors the validation loss during training and saves the model weights to a file (`ner_crf.h5`) whenever the validation loss improves.
- `EarlyStopping`: This callback monitors the validation loss and stops training if there is no improvement after a certain number of epochs (`patience=10`).

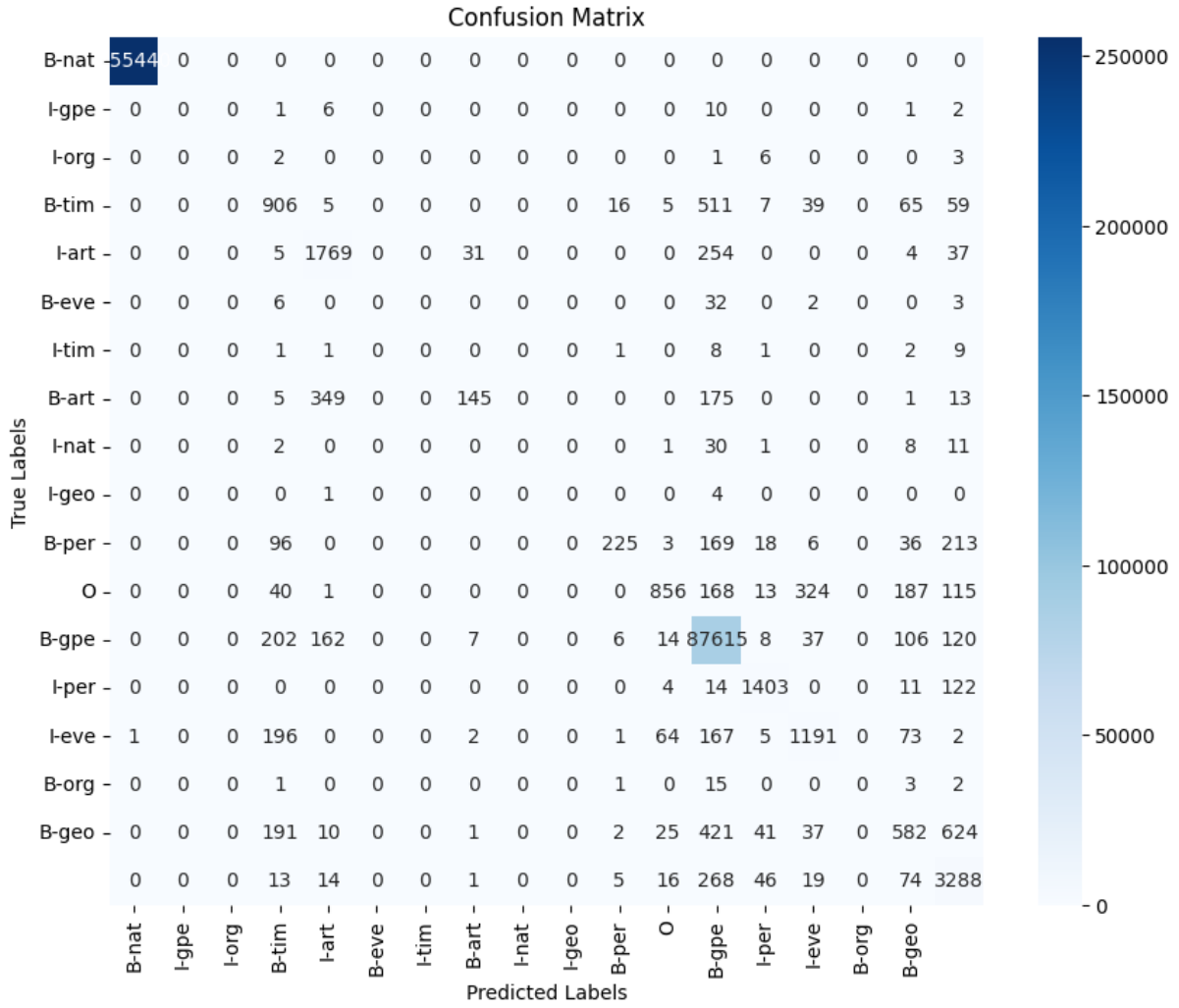
3. **Model Summary**: The summary of the model architecture is printed to the console using `model.summary()`, providing details about the layers, their output shapes, and the number of parameters.
4. **Model Training**:
   - The `fit()` method is called on the model object (`model`) to train the model.
   - Training data (`X_train` and `y_train`) are provided, along with batch size (`batch_size`), number of epochs (`epochs`), validation data (`X_test` and `y_test`), and callbacks (`callbacks`).
   - The model iteratively adjusts its weights during training to minimize the defined loss function (Sigmoid Focal Cross Entropy) based on the provided training data.
5. **Evaluation**:
   - After training, the model's performance is evaluated on the validation data (`X_test` and `y_test`) to assess its generalization capability.
   - Predictions (`y_pred`) are generated for the validation data using the trained model.
   - Predictions are converted back to label indices from one-hot encoded format (`y_pred_labels`) for comparison with the ground truth labels (`y_test_labels`).
   - Token-level metrics such as accuracy are computed using sklearn's `accuracy_score` function by comparing predicted and true labels.
6. **Token-Level Metrics**:
   - The accuracy of the model is calculated as the proportion of correctly predicted labels to the total number of tokens.
   - Other token-level metrics such as precision, recall, and F1-score can also be calculated using `classification_report`, but it is currently commented out.

Overall, this process involves training the model on labeled data, saving the best-performing model, and evaluating its performance on unseen data to assess its effectiveness for the NER task.

Step 5: Evaluation

We've evaluated the test dataset using metrics like accuracy and Confusion matrix. The results are as follows:

```
Accuracy: 0.9825410063942174
```



References

https://huggingface.co/docs/transformers/en/model_doc/bart#transformers.BartTokenizer

https://huggingface.co/docs/transformers/en/model_doc/bart#transformers.BartForConditionalGeneration

https://medium.com/@lidores98/finetuning-huggingface-facebook-bart-model-2c758472e340

https://huggingface.co/docs/transformers/en/tasks/summarization

https://huggingface.co/learn/nlp-course/en/chapter7/5

https://medium.com/@rakeshrajpurohit/customized-evaluation-metrics-with-hugging-face-trainer-3ff00d936f99

https://huggingface.co/docs/transformers/en/main_classes/trainer

https://www.freecodecamp.org/news/what-is-rouge-and-how-it-works-for-evaluation-of-summaries-e059fb8ac840/

https://medium.com/illuin/named-entity-recognition-with-bilstm-cnns-632ba83d3d41

Annotated Corpus for Named Entity Recognition (kaggle.com)

Named Entity Recognition with BiLSTM-CRF and BERT (kaggle.com)

https://towardsdatascience.com/named-entity-recognition-ner-using-keras-bidirectional-lstm-28cd3f301f54