Task 2. Computer Vision. Sentinel-2 Image Matching

Dataset Creation Overview

All the operations for the dataset creation are done in the *"GenerateDatasetAsync"* notebook.
I have created a prompt. The idea is to generate a dataset where each sentence contains a mountain name, and the corresponding tokens are tagged with parts of speech and NER labels like **"B-MOUNTAIN"** and **"I-MOUNTAIN."**

Addressing Repetition in Mountain Names

As the model doesn't track the mountains it generated previously, there was a problem with the repetition of mountain names. For example, 6 out of 10 sentences were generated with the Everest mountain. So I decided to randomly pull one of 60 mountain names from a list of the most popular mountains.

Model Selection

I used the **gpt-4o-mini** model from OpenAI, which is one of the most powerful right now and very easy to use.

Asynchronous Processing for Efficiency

I processed the prompts asynchronously with the *`generate_multiple_responses`* function, so it wasn't waiting on each request to finish before moving to the next one.
To avoid hitting API limits, I set a limit on how many requests are sent at the same time.

Parsing Responses and Storing Data in DataFrame

Next, the goal is to take the generated responses and transform them into a structured dataset. First, I parse the model's response with *`parse_response`*. It extracts the **sentence**, **words**, **POS tags**, and **NER tags** from the generated text. One issue with the responses is that not all tags are valid. For example, the model might include irrelevant tags. To fix this, I check each tag and ensure it fits into one of the allowed categories: **B-MOUNTAIN**, **I-MOUNTAIN**, or **O**. If a tag doesn't belong, it gets replaced with **O**.

Next, the processed data is stored in a list of rows, where each row contains the **sentence**, the **word**, its **POS tag**, and its **NER tag**. This data is then converted into a pandas DataFrame, where I clean up the NER tags by splitting them into general tags (e.g., **MOUNTAIN** or **O**) and positional tags (**B** or **I**).

Formatting

To make sure each word in the sentence is properly aligned with its NER tag, I group the data by sentence and drop any mismatches in length between words and tags. The DataFrame also includes a unique ID for each tag.

I also added some nice formatting—words tagged with **MOUNTAIN** get color-coded for clarity using the display function, *`display_formatted_text`*. This function highlights the mountain names in a visually distinct way when printed.

Saving the Processed Dataset

Finally, the processed dataset is saved to a CSV file for further use, ensuring all the data is neatly structured and ready for analysis or model training.

Model Training

All the operations for the model training and inference are done in the *"NER_Mountains"* notebook.

For the NER task I have decided to use bert-base-cased model.

BERT (Bi-Directional Encoder Representation of transformers), as the name suggests is a transformer model that uses encoder. BERT model has 12 encoders in them.

The input text is converted into tokens and each token is passed as an input to the BERT model independently. There is no time-step dependency in BERT like how it was in LSTMs and GRUs. BERT plainly uses self attention mechanism to understand the context of inputs.

In simple words, attention is a process that determines the importance of the words in the sequence. The words are scored based on importance which helps in context understanding.

Loading BERT Model and Tokenizer

The first step is to load the **BERT model** and its tokenizer:

```
huggingface_model = "bert-base-cased"

tokenizer = AutoTokenizer.from_pretrained(huggingface_model,
do_lower_case=True)
```

The tokenizer is loaded with the `do_lower_case=True` parameter. This ensures that all input text will be converted to lowercase, making it easier to process. Additionally, the tokenizer maps **special tokens** like `[CLS]`, `[SEP]`, and `[PAD]` automatically, which will be important later during training when we need to handle sequence padding.

Exploding and Extracting Tags

Next, we flatten the lists of tags stored in the DataFrame:

```
df_exploded_tag = df.explode('Tag')

df_exploded_tag_general = df.explode('TagGeneral')
```

When you have a DataFrame with lists in the columns, flattening or "exploding" those lists places each tag on a separate row. This way, we can work with each tag individually instead of in a grouped list. Once exploded, unique tags are extracted, and we calculate how many unique tags exist:

```
tag_list = df_exploded_tag["Tag"].unique()

n_tags = len(tag_list)
```

By doing this, we prepare the tags for the next step—mapping them to unique IDs.

Mapping Tags to IDs

For **model training**, tags need to be represented as numerical values rather than strings. The following code maps each tag to a unique ID:

```
tags2ids = {tag: i for i, tag in enumerate(tag_list)}

ids2tags = {i: tag for i, tag in enumerate(tag_list)}
```

This is important because machine learning models work with numbers. For example, the tag `"B-MOUNTAIN"` might be assigned the value `1`, and `"O"` might be assigned `0`. This mapping makes it easy to convert between human-readable tags and model-friendly numeric values.

Model Setup

We then initialize the **BERT model** for token classification:

```
model = BertForTokenClassification.from_pretrained(

    huggingface_model,

    num_labels=n_tags

).to(device)
```

Here, the model is set up to classify tokens, with the number of labels corresponding to the number of unique tags (`n_tags`). The model is moved to the **device**, which is GPU in my case, because the google colab provides it for free. This is where all the training and inference will happen.

Custom Dataset Class

A **CustomDataset** class is defined to handle the input data and token labels. The main purpose of this class is to tokenize the input words and map them to the appropriate labels:

```
class CustomDataset(torch.utils.data.Dataset):

    def __init__(self, df, tokenizer):

        self.inputs = df["Word"].values

        self.labels = df["TagId"].values
```

In the `__getitem__` method, the tokenizer splits the input text into individual tokens and applies the correct labels. Subwords (pieces of words) are ignored in the loss computation by assigning them a special value of `-100`, which tells the model not to calculate loss for those tokens:

```
def __create_token_labels(self, labels, word_ids):

    extended_labels = [-100 if word_id is None else labels[word_id]
for word_id in word_ids]
```

Collate Function for Padding

The **collate_fn** function is responsible for ensuring that all sequences in a batch are padded to the same length. This is important, as BERT requires each input sequence to be the same size:

Dataset Splitting and DataLoader Creation

Once the dataset is ready, we split it into training and validation sets, where training set has 80%(1,163) samples and validation set has 20%(291) samples:

```
train_size = int(0.8 * len(dataset))

train_dataset, validation_dataset = random_split(dataset, [train_size,
validation_size])
```

DataLoaders are created to feed batches of data into the model. The training data is randomly shuffled using `RandomSampler`, while validation data is processed sequentially:

```
train_dataloader = DataLoader(train_dataset,
sampler=RandomSampler(train_dataset), batch_size=batch_size,
collate_fn=collate_fn)

validation_dataloader = DataLoader(validation_dataset,
sampler=SequentialSampler(validation_dataset), batch_size=batch_size,
collate_fn=collate_fn)
```

Training Setup

Optimizers

Optimizers adjust the model's weights to minimize the loss function, which measures how far off the model's predictions are from the actual values. The optimizer uses the gradients (partial derivatives) computed during backpropagation to update these weights.

Schedulers

Schedulers adjust the learning rate during training. A high learning rate at the start helps in quick convergence, while lowering it later on ensures more fine-tuned weight adjustments.

My choice

I tested 3 most popular optimizers such as *AdamW, SGD and RMSprop*. Also I tested 3 scheduler rates such as *CosineAnnealingLR, StepLR* and *ReduceLROnPlateau*. The best duo for this task appears to be **AdamW** optimizer and **CosineAnnealingLR** scheduler:

```
optimizer = AdamW(model.parameters(), lr=2e-5, eps=1e-8)

scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=10)
```

Training Loop

The training loop runs for a specified number of epochs. During each epoch:

1. Gradients are reset with `optimizer.zero_grad()`.
2. The model performs a **forward pass** over the data to predict labels.
3. **Loss** is calculated and backpropagated to update the model's weights.
4. A **progress bar** shows the current training status, including the loss:

```
for epoch_i in range(EPOCHS):

    loop = tqdm(train_dataloader, desc=f"Training. Epoch
[{epoch_i+1}/{EPOCHS}]")
```

After each training epoch, the model is evaluated on the validation set, where no gradients are calculated to save memory and computation time.

Early Stopping and Model Saving

To prevent overfitting, **early stopping** is implemented. If the validation loss does not improve after a few epochs(in my case it is 2 epochs), training is halted and the best model state is saved:

```
if epochs_no_improve >= patience:

    print("Early stopping triggered")

    model.load_state_dict(best_model_state)

    break
```

Evaluating the Model

The `evaluate` function is designed to test the performance of the trained model using the **seqeval** metric that compares its predicted labels for sequences (such as words in a sentence) with the true labels.

```
def evaluate(model, dataloader):

    metric = eval_lib.load("seqeval")
```

The predictions are collected from the model's output logits:

```
output = model(input_ids=batch[0], token_type_ids=batch[1],
attention_mask=batch[2], labels=batch[3])

predictions = torch.argmax(output.logits.detach().cpu(), 2).numpy()
```

The predicted and actual labels are compared, ignoring padding tokens (`-100`).

Seqeval calculates several metrics, including:

- **Precision**: The percentage of correctly predicted entities out of all predicted entities.
- **Recall**: The percentage of correctly predicted entities out of all actual entities in the dataset.
- **F1 Score**: The harmonic mean of precision and recall, providing a single score that balances both.
- **Accuracy**: Measures how often the model correctly predicts the tags.

Example Output:

```
{'MOUNTAIN': {'precision': 0.993, 'recall': 0.984, 'f1': 0.988,
'number': 1730}}
```

This tells us that the model achieved very high precision and recall when predicting mountain names.

Testing the result on an example sentence using hand-made function

Let’s take a sample input text about **Mount Kilimanjaro**:

Mount Kilimanjaro stands as the tallest peak in Africa, drawing adventurers from around the world. Its snow-capped summit contrasts beautifully with the surrounding savannah, offering breathtaking views for those who make the climb. As hikers ascend through its various climate zones, they experience everything from tropical rainforests to alpine deserts. The journey to the top is both physically demanding and mentally challenging, but the reward of standing on the roof of Africa is unforgettable. For many, reaching Kilimanjaro’s summit is a once-in-a-lifetime achievement.

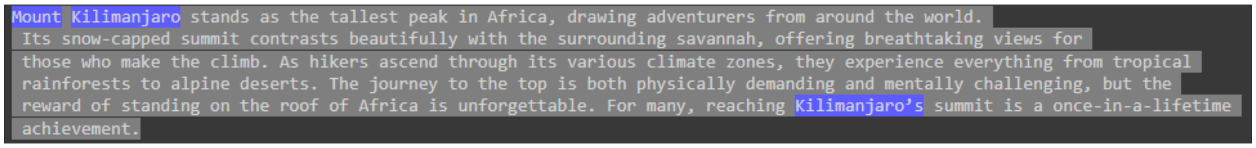
The `predict` function outputs a list of words, along with their predicted tags and confidence scores:

```
[{'word': 'Mount', 'entity_group': 'MOUNTAIN', 'score': 0.9997},

 {'word': 'Kilimanjaro', 'entity_group': 'MOUNTAIN', 'score': 0.9998},

 {'word': 'stands', 'entity_group': 'O', 'score': 0.9999}, ...]
```

This means that the model correctly identified **"Mount"** and **"Kilimanjaro"** as mountains with a confidence of 99.97% and 99.98%, respectively. The word **"stands"** was classified as not being part of an entity (0), with a very high confidence of 99.99%.

Displaying Formatted Text with Entities

If `display_formatted_text=True`, the function also highlights the entity tags directly in the output text. The words classified as **MOUNTAIN** are highlighted in a specific color, making it easier to visualize the predictions:



```
Mount Kilimanjaro stands as the tallest peak in Africa, drawing adventurers from around the world.
 Its snow-capped summit contrasts beautifully with the surrounding savannah, offering breathtaking views for
 those who make the climb. As hikers ascend through its various climate zones, they experience everything from tropical
 rainforests to alpine deserts. The journey to the top is both physically demanding and mentally challenging, but the
 reward of standing on the roof of Africa is unforgettable. For many, reaching Kilimanjaro’s summit is a once-in-a-lifetime
 achievement.
```