

COMP 4211 Group Project Report

Bakhtiyar Temirov, 20902773, Batyrkhan Sakenov, 21010747
btemirov@connect.ust.hk, bsakenov@connect.ust.hk

May 2, 2025

Introduction

In this Named Entity Recognition (NER) project, we fine-tune and compare five transformer-based architectures—base BERT, BERT-Large, RoBERTa, Modern BERT, and DeBERTaV3 on a dataset of 40,000 annotated sentences and evaluate them on a held-out test set of 5,000 examples. Each model is paired with a customized preprocessing pipeline to align word-level entity tags with subword tokenization, ensuring accurate label mapping and efficient batching. We assess their performance using F1-score metrics and resource utilization. Our goal is to determine which architecture offers the best F1-score, given computational resources of free google colab and kaggle GPUs.

The best model we produced was BERT-large-cased, with its parameters listed in the last part of the report and in section on experiments with BERT-large-cased.

Part 0: Description of notebooks in our .zip file

Notebooks folder contains the following notebooks and a folder with the optimal, and selected for submission model.

- **Main file:** bert-large-optimal.ipynb – notebook of training the final model. It is in the folder optimal
- bert-large-notpotimal.ipynb – notebook, containing the grid search of bert-large and full training of suboptimal model
- bert_base - notebook, containing training of bert-base
- roberta.ipynb - notebook of training of roberta model
- modern_bert.ipynb - notebook, where we attempted to train modern bert
- NER_GRID_SEARCH_V3-LARGE.ipynb – notebook, containing grid search for DeBERTaV3 Large
- NER_FINAL_TRAINING_V3-LARGE.ipynb – notebook containing, the full final training for DeBERTaV3 Large using the optimal parameters returned by grid search

Part 1: Data Preprocessing and Exploration

In this section, we describe the detailed steps taken to load, clean, and prepare the Named Entity Recognition (NER) dataset for modeling. Our initial strategy was to prepare the dataset to be used with BERT base model (cased) specifically.

Loading the CSV Files

First, to explore the dataset, the training set contains both sentences and their corresponding NER tags, whereas the test set contains only sentences. Printing the shapes ensures we have the expected number of examples (40,000 for training and 5,000 for test).

```
train_df = pd.read_csv("train.csv")
test_df = pd.read_csv("test.csv")

print("Train shape:", train_df.shape)
print("Test shape: ", test_df.shape)
```

Parsing Stringified Lists

The **Sentence** and **NER Tag** columns are stored as string representations of Python lists. We convert them back to actual lists with `ast.literal_eval`, which safely evaluates each string into a Python list. This step is crucial so that each sentence is a list of tokens and each tag sequence is a list of labels, enabling word-level processing.

```
train_df['Sentence'] = train_df['Sentence'].apply(ast.literal_eval)
train_df['NER Tag'] = train_df['NER Tag'].apply(ast.literal_eval)
test_df['Sentence'] = test_df['Sentence'].apply(ast.literal_eval)
```

Train/Validation Split

To monitor model performance during training, we reserve 10% of the training data as a validation set. Additionally, a fixed `random_state` ensures reproducibility.

```
full_df = train_df.copy()

train_df, val_df = train_test_split(
    train_df,
    test_size=0.1,
    random_state=42
)
```

Label Encoding

Machine learning models require numeric labels. We flatten all tag lists in the training set to find every distinct label (e.g., **B-geo**, **I-org**, **O**) and build `label2id` - a mapping that maps each label to a unique integer, which the model will predict and `id2label` - a mapping that allows conversion back to human-readable tags during evaluation.

```
unique_labels = sorted({
    label
    for tags in train_df['NER Tag']
    for label in tags
})

label2id = { label: i for i, label in enumerate(unique_labels) }
id2label = { i: label for label, i in label2id.items() }
```

Tokenization and Alignment of Labels

We use the `BertTokenizerFast` to tokenize sentences. Since BERT may split words into sub-word tokens, we align the original word-level NER tags to the tokenized inputs by:

1. Assigning the label ID to the first token of each word.
2. Setting labels for subsequent sub-word tokens (and special tokens) to -100, so they are ignored by the loss function.

`BertTokenizerFast` was selected as tokenizer (instead of `AutoTokenizer` or `BertTokenizer`) as it is the fastest one for BERT models.

```

tokenizer = BertTokenizerFast.from_pretrained("bert-base-cased")

def encode_examples(example):
    # Tokenize with word-level alignment
    tokenized_input = tokenizer(..., max_length=128, is_split_into_words=True, ...)

    word_ids = tokenized_input.word_ids(batch_index=0)
    previous_word_idx = None
    label_ids = []

    for word_idx in word_ids:
        if word_idx is None:
            label_ids.append(-100)
        elif word_idx != previous_word_idx:
            label_ids.append(label2id[example["NER Tag"][word_idx]])
        else:
            label_ids.append(-100)
        previous_word_idx = word_idx

    tokenized_input["labels"] = torch.tensor(label_ids)

    return {
        k: v.squeeze() if isinstance(v, torch.Tensor) else v
        for k, v in tokenized_input.items()
    }

```

Details:

- `is_split_into_words=True` informs the tokenizer that input is already split into tokens.
- `word_ids()` provides a mapping from each token back to its original word index.
- We assign `-100` to tokens we wish to ignore in the loss (sub-words and special tokens), following the convention used by HuggingFace’s token classification models.

```

train_dataset = train_dataset.map(
    encode_examples,
    remove_columns=train_dataset.column_names
)
val_dataset = val_dataset.map(
    encode_examples,
    remove_columns=val_dataset.column_names
)

```

At the end of these preprocessing steps, we have:

- **train_dataset** and **val_dataset**, each containing tokenized inputs and aligned label IDs.
- A consistent label encoding scheme via `label2id` and `id2label`.

The outlined data pre-processing pipeline was used to fine-tune the following pre-trained models: base BERT, BERT-Large, RoBERTa, and Modern BERT.

DeBERTaV3 Base: Modified Data Preprocessing Pipeline

The data pre-processing pipeline was slightly modified for the next pre-trained model - DeBERTaV3 Base. While most of the steps remained unchanged, this section outlines some of the key updates.

Model & Tokenizer Initialization

We select the DeBERTaV3 backbone and configure it for token classification.

```
backbone = 'microsoft/deberta-v3-base'
tokenizer = AutoTokenizer.from_pretrained(backbone)
config = AutoConfig.from_pretrained(
    backbone,
    num_labels=len(unique_labels),
    output_hidden_states=False
)

max_length = 128
batch_size = 16
```

Details:

- `AutoTokenizer` and `AutoConfig` automatize loading the correct vocabulary and model settings.
- `num_labels` ensures the model head matches our tag set size.
- `max_length` and `batch_size` are chosen to balance GPU memory and sequence coverage.

Train/Validation Split and DataLoader Setup

We split our `full_ds` into training and validation folds and prepare PyTorch `DataLoaders` with appropriate collators.

```
split = full_ds.train_test_split(test_size=0.2, seed=42)
train_ds = split['train']
val_ds = split['test']

data_collator = DataCollatorForTokenClassification(tokenizer)

train_loader = DataLoader(
    train_ds,
    batch_size=batch_size,
    shuffle=True,
    collate_fn=data_collator
)
val_loader = DataLoader(
    val_ds,
    batch_size=batch_size,
    shuffle=False,
    collate_fn=lambda x: tokenizer.pad(x, return_tensors='pt')
)
test_loader = DataLoader(
    dtest,
    batch_size=batch_size,
    shuffle=False,
    collate_fn=lambda x: tokenizer.pad(x, return_tensors='pt')
)
```

Details:

- We use `Dataset.train_test_split (80/20)` instead of an external splitter.
- `DataCollatorForTokenClassification` handles dynamic padding and creates the attention masks, labels, and token type IDs.
- For validation and test, we manually pad each batch using `tokenizer.pad(...)` to return PyTorch tensors.

The updated pipeline was used to fine-tune DeBERTaV3 Base and DeBERTaV3 Large models.

Part 2: Description of methods and models

Rationale for Pre-trained Language Models

Transformer-based language models such as BERT and its successors learn rich, contextualized token representations by pre-training on massive corpora with masked-language and next-sentence objectives. For NER, these pre-trained representations help the model use syntactic and semantic knowledge—even for rare entities—improving performance with limited labeled data.

As the first model that we tried for this project was BERT-base, which produced $F1 \approx 0.82$. After reading more about other models (i.e. LSTM) used typically NER task, we decided to focus on fine-tuning BERT variations. As a results, we had a big focus on DeBERTa as well.

Model Variants

We compare six Transformer architectures, all adapted for token-level classification:

- **BERT-base** (109M parameters)
- **BERT-Large** (336M parameters)
- **RoBERTa-base** (125M parameters)
- **Modern BERT (base)** (149M parameters)
- **DeBERTaV3-base** (86M parameters)
- **DeBERTaV3-large** (304M parameters)

Regularization

For some of the models BERT-base, BERT-Large, RoBERTa, and Modern BERT we explicitly set standard dropout rates of 0.1 in the token-classification head:

```
model = BertForTokenClassification.from_pretrained(  
    "bert-large-cased",  
    num_labels=len(label2id),  
    id2label=id2label,  
    label2id=label2id,  
    attention_probs_dropout_prob=0.1,  
    hidden_dropout_prob=0.1  
)
```

Details:

- **attention_probs_dropout_prob** applies dropout to attention weights, reducing co-adaptation among heads.
- **hidden_dropout_prob** applies dropout after fully connected layers, reducing the chance of overfitting.

For the DeBERTaV3-base and DeBERTaV3-large models, we rely on their default configuration, which likewise employs dropout rates of 0.1 in both attention and feed-forward sublayers:

```

config = AutoConfig.from_pretrained(
    "microsoft/deberta-v3-base",
    num_labels=len(label2id),
    output_hidden_states=False
)

```

Classification Head

DeBERTaV3 models learn a linear mapping from the final hidden states of each token to the NER tag logits:

```

self.classifier = nn.Linear(
    config.hidden_size,
    config.num_labels
)

```

The `nn.Linear` layer projects the contextualized representation of each token (of dimension `hidden_size`) into a logit vector over the set of NER labels, enabling per-token classification CRF.

CRF Layer for DeBERTaV3 Models

To better capture valid IOB tag transitions and enforce sequence-level consistency, we augment the DeBERTaV3 models with a Conditional Random Field (CRF) on top of the token logits:

```

class DebertaCRF(nn.Module):
    def __init__(self, config):
        ...
        self.crf = CRF(config.num_labels, batch_first=True)
        ...

```

Advantages of CRF:

- **Global sequence modeling:** CRF maximizes the joint probability of the entire tag sequence, rather than independent token-wise probabilities.
- **IOB constraint enforcement:** Transition scores in the CRF can learn to penalize illegal tag sequences (e.g., I-PER following O), improving F1.
- **Better boundary detection:** By modeling dependencies between adjacent tags, CRF often yields more accurate entity span predictions than plain softmax classification.

Part 3: Hyperparameter Tuning Methodology

To optimize the performance of our models, we conducted a grid search over a set of key hyperparameters using the `ParameterGrid` utility from `sklearn.model_selection`. Here is an example of parameter grid for `bert-large-cased`:

```

param_grid = {
    'learning_rate': [2e-5, 6e-5, 1e-4],
    'per_device_train_batch_size': [8, 16],
    'num_train_epochs': [2],
    'weight_decay': [0.01]
}

best_f1 = 0
best_params = {}

for params in ParameterGrid(param_grid):
    print(f"Training with parameters: {params}")
    model_dir = f"model_lr_{params['learning_rate']}_bs_{params['per_device_train_batch_size']}_epochs_{params['num_train_epochs']}_wd_{params['weight_decay']}"
    training_args = TrainingArguments(
        output_dir=f"results",
        eval_strategy="epoch",
        save_strategy="epoch",
        learning_rate=params['learning_rate'],
        per_device_train_batch_size=params['per_device_train_batch_size'],
        per_device_eval_batch_size=16, #evaluation batch size consistent
        num_train_epochs=params['num_train_epochs'],
        weight_decay=params['weight_decay'],
        save_total_limit=1,
        push_to_hub=False,
        load_best_model_at_end=True,
        metric_for_best_model="f1" #use f1 as metric
    )

```

Parameter grid for bert-large-cased

Due to the limited computational resources, we focused on tuning the most impactful hyperparameters - learning rate and train batch size with 2 epochs, while keeping weight decay in its often used value 0.001. Then we would use most optimal parameters and train the corresponding model for 3 epochs to generate submission files. For all variations of BERT, we set most commonly used learning rates that ranged from $1 \cdot 10^{-5}$ to $1 \cdot 10^{-4}$ and batch sizes of 8, 16, 32 (depending if GPU could handle more batch sizes).

Also, for efficiency we performed training in grid search not on the full set of training samples, but on randomly chosen 20% of them. Accordingly, we used 20% of the validation set for validation. Those smaller training sets and validation sets were defined as train_tuning and val_tuning

```

# Split into train/validation (adjust test_size as preferred)
full_df = train_df # saving the full dataset for further training
train_df, val_df = train_test_split(train_df, test_size=0.1, random_state=42)

# sets used for hyperparameter tuning (20% of training set and validation set)
train_tuning, remaining_test = train_test_split(train_df, test_size=0.8, random_state=42)
val_tuning, remaining_validation = train_test_split(val_df, test_size=0.8, random_state=42)

```

Defining smaller sets for tuning

The training process for each configuration included saving the model weights and monitoring the F1 score on smaller validation set. After training, we compared the F1 scores across all configurations to identify the best-performing hyperparameters.

a reduced “tuning” dataset, then exhaustively trained on each combination of hyperparameters, and finally selected the set yielding the highest validation F1 score.

Tuning Set Construction

We reserve 10% of the original training set for quick hyperparameter experiments. This small split allows rapid iteration at the cost of lower absolute performance.

```
tune_df, _ = train_test_split(train_df,
                              test_size=0.9,
                              random_state=42)
tune_train_df, tune_val_df = train_test_split(
    tune_df,
    test_size=0.2,
    random_state=42
)
...
tune_train_loader = DataLoader(
    tune_train_ds,
    batch_size=8,
    shuffle=True,
    collate_fn=data_collator
)
tune_val_loader = DataLoader(
    tune_val_ds,
    batch_size=8,
    shuffle=False,
    collate_fn=data_collator
)
...
```

Details:

- We sample 10% of the original `train_df`, then split that into 80/20 train/validation. This yields roughly 3,200 training and 800 validation examples—small enough for fast runs.
- We reuse our existing `tokenize_and_align_labels` and `remove_word_ids` functions to produce ready-to-train `Datasets`.
- A fixed batch size of 8 ensures each experiment uses minimal GPU memory and that changing batch size will be straightforward in the grid search.

Grid Search over Hyperparameters

We define the search space for learning rate, batch size, and number of epochs, then iterate through every combination.

```
param_grid = {
    "lr": [1e-5, 5e-5],
    "batch_size": [8, 16],
    "num_epochs": [1, 3],
}
best = {"f1": 0.0, "params": None}

for params in ParameterGrid(param_grid):
    bt = params["batch_size"]
    ...
    m = DeBERTaCRF(config).to(device)
    m.bert.gradient_checkpointing_enable()
    opt = AdamW(m.parameters(),
                lr=params["lr"],
                weight_decay=0.01)
    total_steps = len(tune_train_loader) * params["num_epochs"]
    sched = get_linear_schedule_with_warmup(
        opt,
        int(0.1 * total_steps),
        total_steps
    )
    scaler = torch.amp.GradScaler()
```

```

for epoch in range(params["num_epochs"]):
    m.train()
    for b in tqdm(tune_train_loader,
                  desc=f"Epoch {epoch} train"):
        ids = b["input_ids"].to(device)
        mask = b["attention_mask"].to(device)
        lbls = b["labels"].to(device)
        opt.zero_grad()
        with torch.amp.autocast(device_type="cuda"):
            loss, _ = m(ids, mask, lbls)
        scaler.scale(loss).backward()
        scaler.unscale_(opt)
        torch.nn.utils.clip_grad_norm_(m.parameters(),
                                       1.0)

        scaler.step(opt)
        scaler.update()
        sched.step()

m.eval()
true_seq, pred_seq = [], []
with torch.no_grad():
    for b in tune_val_loader:
        ids = b["input_ids"].to(device)
        mask = b["attention_mask"].to(device)
        lbls = b["labels"].cpu().numpy()
        preds = m(ids, mask)
        for p, l, msk in zip(preds,
                              lbls,
                              b["attention_mask"].numpy()):
            t = [id2label[tag]
                  for tag, mk in zip(l, msk)
                  if mk and tag != -100]
            pr = [id2label[tag]
                  for tag, mk, lt
                  in zip(p, msk, l)
                  if mk and lt != -100]
            true_seq.append(t)
            pred_seq.append(pr)
f1 = f1_score(true_seq, pred_seq)

if f1 > best["f1"]:
    best.update({"f1": f1, "params": params})

```

Details:

- ParameterGrid builds all 8 combinations of {lr, batch_size, num_epochs}.
- For each tuple params, we rebuild DataLoaders with the new batch size, reinitialize the model, optimizer, scheduler (10% warmup), and mixed-precision scaler.
- We clip gradients to a norm of 1.0 to stabilize training on the small tuning set.
- After training for the specified number of epochs, we compute validation F1 using the same decode logic as in full training.
- We track the best F1 and its associated hyperparameters.

Part 4: Experiments with different versions of BERT

Base BERT

Base BERT was the first model we tried and produced f1 around 0.857 for submission. First base-bert was trained with 3 epochs and the following settings.

```
[ ] # Set up the model
model = BertForTokenClassification.from_pretrained(
    "bert-base-cased",
    num_labels=len(label2id),
    id2label=id2label,
    label2id=label2id,
    attention_probs_dropout_prob=0.1,
    hidden_dropout_prob=0.1
)
```

BERT-base-cased general settings

```
# Set training arguments
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

# Set up the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer)
```

BERT-base-cased general hyperparameters



submission_dropout.csv

Complete · Batyrkhan Sakenov · 3d ago

Score: 0.857557

Results on submission for BERT base with 3 epochs

Then it was trained with 6 epochs and produced slightly better results on the validation set and 0.858 on the submission. It could be seen that increasing number of epochs improves f1 score.

[13500/13500 1:21:48, Epoch 6/6]

| Epoch | Training Loss | Validation Loss | F1 |
|-------|---------------|-----------------|----------|
| 1 | 0.101200 | 0.092147 | 0.830301 |
| 2 | 0.077200 | 0.086476 | 0.832721 |
| 3 | 0.060000 | 0.087809 | 0.840006 |
| 4 | 0.046700 | 0.095723 | 0.840032 |
| 5 | 0.035200 | 0.102781 | 0.841566 |
| 6 | 0.027700 | 0.108330 | 0.842593 |

Results on validation set of BERT base with 6 epochs



submission_regular_epoch6.csv
Complete · Batyrkhan Sakenov · 3d ago

0.858656



Results of submission of BERT base with 6 epochs

Changes of the learning rate (from $2 \cdot 10^{-5}$ to $1 \cdot 10^{-4}$) and batch size (from 16 to 32) also led to faster convergence and slightly better f1 score on validation, but on submission and not enough to see substantial improvements. Hyperparameters were set after applying a gridsearch from part 3.



submission_higherlearningrate32.csv

Complete · Batyrkhan Sakenov · 3d ago

Score: 0.857923

BERT base with tuned learning rate and batch size

Our best version of bert-base did not breach the bar of $f1 = 0.86$, so we decided to focus on other variants of BERT.

Large BERT

Large BERT was first trained on the same hyperparameters as base-bert, but without droupout and already produced better results than base.

```
model = BertForTokenClassification.from_pretrained(  
    "bert-large-cased",  
    num_labels=len(label2id),  
    id2label=id2label,  
    label2id=label2id  
)
```

Settings of first large BERT

```
training_args = TrainingArguments(  
    output_dir="./results",  
    eval_strategy="epoch",  
    save_strategy="epoch",  
    learning_rate=2e-5,  
    per_device_train_batch_size=16,  
    per_device_eval_batch_size=16,  
    num_train_epochs=3,  
    weight_decay=0.01,  
    save_total_limit=1,  
    push_to_hub=False,  
)  
  
trainer = Trainer(  
    model=model,  
    args=training_args,  
    train_dataset=train_dataset,  
    eval_dataset=val_dataset,  
    compute_metrics=compute_metrics,  
    tokenizer=tokenizer,  
    callbacks=[SaveCheckpointCallback]  
)
```

Hyperparameters of initial large BERT

```
[ ] # Train!
    trainer.train()

# Save the model to a local directory (e.g., 'model')
trainer.save_model("bert_large_model")
```

[6750/6750 2:14:18, Epoch 3/3]

| Epoch | Training Loss | Validation Loss | F1 |
|-------|---------------|-----------------|----------|
| 1 | 0.099500 | 0.090156 | 0.833578 |
| 2 | 0.070200 | 0.083333 | 0.836795 |
| 3 | 0.048600 | 0.086929 | 0.845771 |

Results on validation of first BERT

submission_large_bert.csv
Complete · Batyrkhan Sakenov · 3d ago

0.862052

Results of submission of first BERT

After that, hyperparameter tuning was applied and resulted in best learning rate being equal to $6 \cdot 10^{-5}$ and batch size 16. We decided to train it for 4 epochs instead of 3 to improve performance.

```
# Set training arguments
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=6e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=4,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

# Set up the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer
)
```

Hyperparameters of first optimized large BERT, with 4 epochs

However, it produced lower $F1$ on the third epoch and worse performance on submission.

[9000/9000 2:53:29, Epoch 4/4]

| Epoch | Training Loss | Validation Loss | F1 |
|-------|---------------|-----------------|----------|
| 1 | 0.099900 | 0.091133 | 0.826858 |
| 2 | 0.068800 | 0.086657 | 0.835594 |
| 3 | 0.039300 | 0.096981 | 0.844492 |
| 4 | 0.017900 | 0.118275 | 0.848465 |

Validation results of optimized large BERT

submission_final.csv
Complete · Batyrkhan Sakenov · 1d ago

0.856396

Validation results of optimized large BERT

This $F1$ turned out to be less than that of the initial bert-large and even bert-base. A possible reason could be that learning rate $6 \cdot 10^{-5}$ was best for 2 epochs and that particular subset of training data, but worse for 3 epochs and full training set than $2 \cdot 10^{-5}$. Another reason for drop of performance, could have been that by training our model for 4 epochs, it got overfitted to the training data and thus performed worse on unseen data of the public test. Therefore, we decided to go back to the original settings of our bert-large with learning rate $2 \cdot 10^{-5}$, batch size 16 and the performance 0.862 on the public test as we believe it was less overfit for the test data. Therefore, the final hyperparameters of our best bert-large are

```
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer,
    callbacks=[SaveCheckpointCallback]
)
```

Hyperparameters of our Final submission.

RoBERTa

Roberta-base, with the following parameters was trained with 3 epochs and produced our worst results, around $F1 = 0.75$, on submission and low $F1$ on validation set.

```
# Set up the model
model = RobertaForMaskedLM.from_pretrained(
    "roberta-base",
    num_labels=len(label2id),
    id2label=id2label,
    label2id=label2id,
    attention_probs_dropout_prob=0.1,
    hidden_dropout_prob=0.1
)
```

Settings of RoBERTa-base

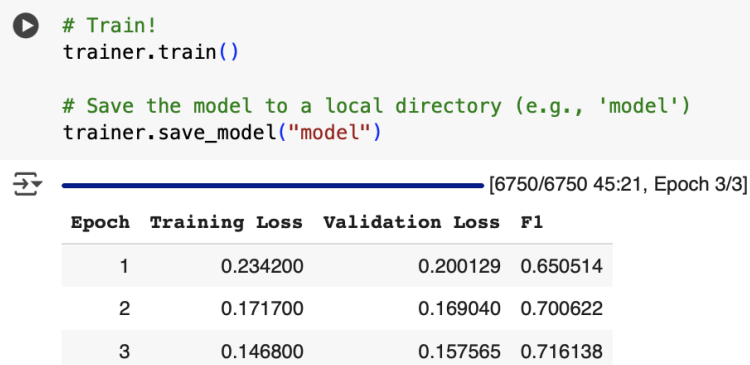
```

# Set training arguments
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

# Set up the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer
)

```

Hyperparameters of RoBERTa-base



Results on validation of RoBERTa-base after 3 epochs



Submission results of RoBERTa-base

Even though, its performance increased from epoch 1 to epoch 2 significantly, it did not increase from epoch 2 to epoch 3, which likely meant that the model would converge in the range of epochs 2 and 3 and their $f1 \approx 0.75$, even with the optimal hyperparameters. This result was significantly worse than that of original BERT variations and therefore we decided not to proceed with further experimentation on RoBERTa.

Modern BERT

We tried to use the variation of BERT called Modern BERT, from the following source: Modern BERT huggingface.


```

# Set up the model
model = AutoModelForMaskedLM.from_pretrained(
    "answerdotai/ModernBERT-base",
    num_labels=len(label2id),
    id2label=id2label,
    label2id=label2id,
    attention_probs_dropout_prob=0.1,
    hidden_dropout_prob=0.1
)

```

config.json: 100% 1.19k/1.19k [00:00<00:00, 118kB/s]

model.safetensors: 100% 599M/599M [00:02<00:00, 254MB/s]

Settings of Modern Bert

```

# Set training arguments
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=4,
    num_train_epochs=1,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

# Set up the Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer
)

```

Settings of Modern Bert

However, the computing resources of free version of google colab were not enough despite changes in batch sizes (from 16 to 8 for training and from 16 to 4 for evaluation) as CUDA consistently ran out of memory. Even after enabling expandable memory segments for CUDA, out of memory error still occurred.

```

import os
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
import torch

```

Attempting to allocate more memory from pytorch

```

import os
os.environ['PYTORCH_CUDA_ALLOC_CONF'] = 'expandable_segments:True'
import torch

```

Attempting to allocate more memory from pytorch

```

--> 88 return torch.cat((tensor1, tensor2), dim=0)
89
90 # Let's figure out the new shape

```

OutOfMemoryError: CUDA out of memory. Tried to allocate 4.42 GiB. GPU 0 has a total capacity of 14.74 GiB of which 4.42 GiB is free. Process 2505 has 10.31 GiB memory in use. Of the allocated memory 9.91 GiB is allocated by PyTorch, and 277.16 MiB is reserved by PyTorch but unallocated. If reserved but unallocated memory is large try setting PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True to avoid fragmentation. See documentation for Memory Management (<https://pytorch.org/docs/stable/notes/cuda.html#environment-variables>)

Persisting error of CUDA out of memory

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 4.42 GiB.
GPU 0 has a total capacity of 14.74 GiB of which 4.42 GiB is free.
Process 2505 has 10.31 GiB memory in use. Of the allocated memory 9.91GiB
is allocated by PyTorch, and 277.16 MiB is reserved by PyTorch
but unallocated.
```

Our computing resources of free colab were not sufficient for fine-tuning this model, so we focused on DeBERTaV3.

DeBERTaV3

We present our initial fine-tuning experiments with the DeBERTaV3-base and DeBERTaV3-large models. We use the same DebertaCRF architecture for both, varying only the training hyperparameters.

DeBERTaV3-base Initial Training

Model Definition

```
class DebertaCRF(nn.Module):
    def __init__(self, config):
        ...
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        self.crf = CRF(config.num_labels, batch_first=True)

    def forward(self, input_ids, attention_mask, labels=None):
        h = self.bert(input_ids=input_ids,
                      attention_mask=attention_mask)
        .last_hidden_state
        logits = self.classifier(h)
        if labels is not None:
            lab = labels.clone().masked_fill(labels < 0, 0)
            loss = -self.crf(logits, lab, mask=attention_mask.bool(),
                             reduction='mean')
            preds = self.crf.decode(logits, mask=attention_mask.bool())
            return loss, preds
        else:
            return self.crf.decode(logits, mask=attention_mask.bool())
```

Details:

- `AutoModel.from_config` loads the DeBERTaV3 transformer encoder without a downstream head.
- A linear classifier maps each token's hidden state (dimension `hidden_size`) to `num_labels`.
- The CRF layer enforces valid IOB tag transitions and computes a sequence-level log-likelihood.
- During training, we mask out special tokens and subword continuations (`labels < 0`) and optimize the negative log-likelihood.

Training Hyperparameters

```
num_epochs = 3
lr = 3e-5
patience = 2
```

Details:

- num_epochs=3: balances sufficient learning with compute budget.
- lr=3e-5: a standard fine-tuning rate for large pre-trained models.
- patience=2: stops training if validation F1 does not improve for two consecutive epochs.

Training Loop

```

model = DebertaCRF(config).to(device)
model.bert.gradient_checkpointing_enable()
optimizer = AdamW(model.parameters(), lr=lr)
total_steps = len(train_loader) * num_epochs
scheduler = get_linear_schedule_with_warmup(
    optimizer,
    int(0.1 * total_steps), # 10% warmup
    total_steps
)
scaler = torch.amp.GradScaler()

best_f1, patience_ctr = 0.0, 0

for epoch in range(num_epochs):
    model.train()
    train_losses = []
    for batch in tqdm(train_loader, desc=f"Epoch {epoch} train"):
        ids = batch['input_ids'].to(device)
        mask = batch['attention_mask'].to(device)
        lbls = batch['labels'].to(device)
        optimizer.zero_grad()
        with torch.amp.autocast(device_type='cuda'):
            loss, _ = model(ids, mask, lbls)
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()
        train_losses.append(loss.item())
    print(f"Epoch {epoch} avg loss: {sum(train_losses)/len(train_losses):.4f}")

    # --- Validation ---
    model.eval()
    true_seq, pred_seq = [], []
    with torch.no_grad():
        for batch in val_loader:
            ids = batch['input_ids'].to(device)
            mask = batch['attention_mask'].to(device)
            lbls = batch['labels'].cpu().numpy()
            preds = model(ids, mask)
            for p, l, m in zip(preds, lbls, batch['attention_mask'].numpy()):
                t = [id2label[tag] for tag, mk in zip(l, m)
                    if mk and tag != -100]
                pr = [id2label[p_tag] for p_tag, mk, true_tag
                    in zip(p, m, l)
                    if mk and true_tag != -100]
                true_seq.append(t)
                pred_seq.append(pr)
    f1 = f1_score(true_seq, pred_seq)
    print(f"Validation F1: {f1:.4f}")

    if f1 > best_f1:
        best_f1, best_state, patience_ctr = f1, model.state_dict(), 0
    else:
        patience_ctr += 1
        if patience_ctr > patience:
            print("Early stopping")
            break

torch.save(best_state, 'best_model_base.pt')
print(f"Best validation F1: {best_f1:.4f}")

```

Explanation of training details:

- gradient_checkpointing_enable() trades compute for memory by re-computing activations in the backward pass.
- AdamW optimizer with weight decay built in.

- `get_linear_schedule_with_warmup`: 10% of total steps are used to linearly ramp up the learning rate, then decay linearly.
- `torch.amp`: automatic mixed precision to speed up training on GPU.
- At each epoch, we compute average training loss and validation F1; we checkpoint the model whenever F1 improves.

DeBERTaV3-base Training Results

- **Epoch 0**: avg loss = 7.0858, validation F1 = 0.7729
- **Epoch 1**: avg loss = 2.3457, validation F1 = 0.8080
- **Epoch 2**: avg loss = 1.5456, validation F1 = 0.8172
- **Best validation F1**: 0.8172

DeBERTaV3-large Initial Training

The DeBERTaV3-large experiments use the identical `DebertaCRF` model and training loop, differing only in the hyperparameters below.

Training Hyperparameters

```
num_epochs = 2 # Fewer epochs due to larger model size
lr = 2e-5 # Slightly lower learning rate
patience = 3 # More patience for convergence
```

DeBERTaV3-large Training Results

- **Epoch 0**: avg loss = 9.9563, validation F1 = 0.7246
- **Epoch 1**: avg loss = 3.0958, validation F1 = 0.7917
- **Best validation F1**: 0.7917

These initial experiments show that DeBERTaV3-base achieved higher peak F1 (0.8172) with more stable training losses, while DeBERTaV3-large—despite greater capacity—required more careful tuning to match or exceed the smaller variant. The following are the models’ public scores on Kaggle:

- **DeBERTa-v3-base** achieved an F_1 score of 0.840745 on the public leaderboard.
- **DeBERTa-v3-large** improved to an F_1 score of 0.841937 after initial training.



| | | | |
|---|--|-----------------|--------------------------|
|  | solutions-5.csv Complete · qasqyr · 5h ago | 0.841937 | <input type="checkbox"/> |
|  | solutions-4.csv Complete · qasqyr · 6h ago | 0.840745 | <input type="checkbox"/> |

Figure 1: Public leaderboard scores

DeBERTa-v3-large Best Hyperparameters

```
print("Best tuning F1:", best["f1"])
print("Best hyperparameters:", best["params"])
# → Best tuning F1: 0.6725286437516653
# → Best hyperparameters: {'batch_size': 16,
#                           'lr': 5e-05,
#                           'num_epochs': 3}
```

Tuning Results (per combination):

- {bs=8, lr=1e-5, epochs=1} → F1 = 0.5042
- {8, 1e-5, 3} → F1 = 0.6179
- {8, 5e-5, 1} → F1 = 0.5602
- {8, 5e-5, 3} → F1 = 0.6590
- {16, 1e-5, 1} → F1 = 0.4306
- {16, 1e-5, 3} → F1 = 0.5669
- {16, 5e-5, 1} → F1 = 0.4052
- {16, 5e-5, 3} → F1 = 0.6725

Discussion:

- The best tuning F1 (0.6725) was achieved with `batch_size=16`, `lr=5e-5`, and `num_epochs=3`.
- Larger batch size (16) consistently outperformed batch size 8, likely due to more stable gradient estimates.
- A higher learning rate (5e-5) yielded better adaptation in just three epochs compared to 1e-5.
- Training for three epochs improved F1 over a single epoch in all cases.
- The absolute F1 scores on this small tuning subset are lower than those on the full validation set (0.79–0.82). This is expected because:
 - The tuning set contains only a fraction of the data, limiting the model’s ability to generalize.
 - Shorter training (max 3 epochs) and noisier gradient estimates on small batches further reduce ultimate F1.
 - We use early stopping and no data augmentation, so the model may underfit this reduced dataset.

With these hyperparameters identified, we proceed to full-scale training of DeBERTaV3-large using `lr=5e-5`, `batch_size=16`, and a larger `num_epochs=7`.

DeBERTaV3-large Final Training

Training Hyperparameters

For our full-scale DeBERTaV3-large fine-tuning, we used:

- **Learning rate (lr)** = 5×10^{-5} : step size for each optimizer update.
- **Batch size** = **16**: number of examples per gradient computation.
- **Number of epochs** = **7**: full passes over the entire training set.
- **Patience** = **2**: early-stopping rounds without F_1 improvement before halting.

Validation Performance

- **Epoch 0**: avg loss = 8.2463, val F_1 = 0.7565
- **Epoch 1**: avg loss = 2.8330, val F_1 = 0.7702
- **Epoch 2**: avg loss = 2.1752, val F_1 = 0.7988
- **Epoch 3**: avg loss = 1.7922, val F_1 = 0.8105
- **Epoch 4**: avg loss = 1.3939, val F_1 = 0.8128
- **Epoch 5**: avg loss = 1.0053, val F_1 = 0.8195
- **Epoch 6**: avg loss = 0.6534, val F_1 = 0.8228
- **Best validation F_1** : 0.8228 (Epoch 6, triggered patience = 2)

Public Leaderboard Performance

On the Kaggle public leaderboard, the final / full-data model achieved

$$F_1^{\text{public}} = 0.842426,$$

a very slight improvement over the initial DeBERTaV3-large result (0.841937).



| Submission and Description | | Public Score  | Select |
|---|--|--|--------------------------|
|  | solutions (1).csv Complete · qasqyr · 9h ago | 0.842426 | <input type="checkbox"/> |

Figure 2: Public leaderboard score

Comparison to Initial Training

- **Initial setup**: 2 epochs, lr = $2e-5$, patience = 3 \rightarrow best val F_1 = 0.7917.
- **Final setup**: 7 epochs, lr = $5e-5$, patience = 2 \rightarrow best val F_1 = 0.8228.
- **Gain in validation F_1 (+0.0311)** reflects the benefit of a higher learning rate and longer training under early stopping.
- **Public F_1 gain (+0.0005)** demonstrates robustness: small but consistent improvement under the full training loop.

Results

- Increasing epochs allowed the model to refine entity boundary decisions, as evidenced by steadily decreasing training loss and rising F_1 through Epoch 6.
- A larger learning rate accelerated convergence on the full dataset, though early stopping prevented overfitting.
- The modest public leaderboard improvement indicates diminishing returns at this model scale—further gains may require architectural changes or additional data.
- However, our best DeBERTa still achieved results lower F_1 —score than our BERT-Large models.

Part 5: Discussion of Results

Our best models achieved maximum results of in the range 0.81 – 0.85 on validation set and 0.81 – 0.86 on the test set.

Our best submission result was achieved by BERT-large-cased with the public score 0.862 with the following parameters.

```
model = BertForTokenClassification.from_pretrained(
    "bert-large-cased",
    num_labels=len(label2id),
    id2label=id2label,
    label2id=label2id
)

training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    save_total_limit=1,
    push_to_hub=False,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    compute_metrics=compute_metrics,
    tokenizer=tokenizer
)
```

Overall, the original versions of BERT (base, large) performed better than its other variants such as RoBERTa and DeBERTa. This is likely due to RoBERTa and DeBERTa using different encoding technique (byte-level Byte-Pair-Encoding) and being trained on different data than original BERT.

Bert-base had slightly worse performance than its large variation. We think this can be attributed to large BERT having 3 times more parameters and more attention heads, meaning it could capture more long-range relationships between distant words in the sentences (better long-term memory).

A potential point for improvement, is performing grid search on 3 epochs on full dataset instead of 20% of it, to find optimal parameters.

Another potential point for a slight improvement of large-bert performance could have been checking more learning rates and batch sizes, but improvements in $F1$ are likely to be not too significant.

We think the reason why our models could not achieve $f1$ score higher than 0.87 was due to class imbalances and perhaps having too few samples of one class for our models to learn to classify it correctly.

Division of labor in teamwork:

50 : 50