


Fine-Tuning of Large Language Models on Twitter Sentiment Analysis: A Comparative Study of Layer Unfreezing, Hyperparameter Search and Prompt Tuning

Abstract

Fine-tuning large language models (LLMs) for domain-specific tasks, such as sentiment analysis on social media platforms like Twitter. This study explores the effectiveness of three fine-tuning techniques: unfreezing layers, hyperparameter search, and prompt tuning applied to a DistilBERT model. The dataset, consisting of tweets labeled as positive, negative, or neutral, underwent extensive preprocessing to prepare it for model training and evaluating.

Data collection

Let's start with [dataset](#) that was used for this research. That was Twitter Sentiment Analysis.

# 2401 Tweet ID	Borderlands entity	Positive sentiment	im getting on bor... Tweet content
 1 13.2k	Microsoft 3% TomClancysRainb... 3% Other (69881) 94%	Negative 30% Positive 28% Other (31308) 42%	69491 unique values
2401	Borderlands	Positive	I am coming to the borders and I will kill you all,
2401	Borderlands	Positive	im getting on borderlands and i will kill you all,
2401	Borderlands	Positive	im coming on borderlands and i will murder you all,

It has 4 dimensions, but only 2 were used for training and evaluating: Sentiment and Tweet content. Other 2 columns can be used for additional EDA, but it isn't point of this work.

Here you can see example of tweet posts:

```
2403,Borderlands,Neutral,"Rock-Hard La Varlope, RARE & POWERFUL, HANDSOME JACKPOT, Borderlands 3 (Xbox) dlvr.it/RMTrgF"
2403,Borderlands,Neutral,"Rock-Hard La Varlope, RARE & POWERFUL, HANDSOME JACKPOT, Borderlands 3 (Xbox) dlvr.it / RMTrgF"
2403,Borderlands,Neutral,"Rock-Hard La Varlope, RARE & POWERFUL, HANDSOME JACKPOT, Borderlands 3 (Xbox) dfr.it / RMTrgF"
2403,Borderlands,Neutral,"Rock-Hard La Vita, RARE BUT POWERFUL, HANDSOME JACKPOT, Borderlands 1 (Xbox) dlvr.it/RMTrgF"
2403,Borderlands,Neutral,"Live Rock - Hard music La la Varlope, RARE & the POWERFUL, Live HANDSOME i JACKPOT, Borderlands 3 ( Sega Xbox ) dlvr. From it / e RMTrgF"
2403,Borderlands,Neutral,"I-Hard like me, RARE LONDON DE, HANDSOME 2011, Borderlands 3 (Xbox) dlvr.it/RMTrgF"
```

It's important to pay attention that there are 4 options for sentiment: positive, negative, neutral and irrelevant. For example, we can get unrelated tweet post for entity Borderlands which contains: Appreciate the (sonic) concepts / praxis Valenzuela and Landa-Posas thread together in this talk: multimodal listening, soundwalks, frameworks, participatory action research, and testimonios. . . So many thoughtful and resonant intersections here.. .

So as it can decrease model performance, all such irrelevant tweets were filtered out during the preprocessing stage. Also as Tweet id and entity columns don't play any role for sentiment analysis, it was dropped. Initially, all tweets in the dataset were ordered based on the entity they were associated with, you can see it in above example, which could lead to biases during model training if it is not handled properly. To address this, shuffling process was implemented to randomize the order of the tweets. Twitter content and sentiment columns were renamed to post and label:

```
df = df[df['sentiment'] != 'Irrelevant']
df = df.drop(columns=['Tweet id', 'entity'])
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
```

After filtering out irrelevant tweets, we moved to the next step of text preprocessing, which plays a critical role in preparing the dataset for effective model training. The process involved several steps to clean and standardize text data from column Tweet content.

First of all, tweets were converted to lowercase. It helps to decrease size of vocabulary and DistilBERT uncased model was used that is designed to treat text in a case-insensitive manner

Next, special characters, punctuation, URLs, HTML tags, and other non-alphanumeric entities, such as emojis, were also stripped from the text. This step is essential because such characters generally do not contribute to the semantic meaning of the text in the context of sentiment analysis.

```
text = str(text).lower()
text = re.sub(r'\W', ' ', text)
text = re.sub(r'\s+', ' ', text)
text = re.sub(r'\d', '', text)
text = text.encode('ascii', 'ignore').decode('ascii')
text = re.sub(r'http\S+', '', text)
text = re.sub(r'#\w+', '', text)
text = re.sub(r'<.*?>', '', text)
text = re.sub(r'[\^\\w\\s]', '', text)
```

The text was further processed to remove common stopwords—words that are frequently used in language but carry little semantic weight, such as "and," "the," and "is."

```
def remove_stopwords(text: str) -> str:
    stop_words = set(stopwords.words('english'))
    words = text.split()
    words = [word for word in words if word not in stop_words]
    return ' '.join(words)
```

Lemmatization was applied to reduce words to their base or root form. For instance, words like “running,” “ran,” and “runner” would all be reduced to “run.” This normalization process helps the model recognize different forms of a word as the same entity, which enhances its ability to learn and generalize from the data.

```
def lemmatize(text: str) -> str:
    lemmatizer = WordNetLemmatizer()
    words = text.split()
    words = [lemmatizer.lemmatize(word) for word in words]
    return ' '.join(words)
```

After this text preprocessing all models show quite poor performance. So after few experiments it was decided not to perform lemmatization and removing stopwords. These techniques led to reducing of words in twitter posts, and it was reason why model couldn't to classify them properly. So this techniques were not used on final data for training and evaluating.

These preprocessing steps collectively improved the quality of the input data, ensuring that the model was trained on text that is clean.

Also before feeding data into model, it was converted into Hugging Face Dataset objects for more efficient data handling.

```
train_dataset = Dataset.from_pandas(df_train)
val_dataset = Dataset.from_pandas(df_test)
```

After that we need to tokenize the text data and format it for training:

```
def tokenize_data(tokenizer: DistilBertTokenizerFast, dataset: Dataset, prompt: str = None) -> Dataset:

    def tokenize(examples):
        return tokenizer(examples['post'], padding='max_length',
truncation=True, max_length=128)

    dataset = dataset.map(tokenize, batched=True)
    dataset.set_format(type='torch', columns=['input_ids', 'attention_mask', 'label'])

    return dataset
```

It uses the tokenizer to convert the text data from the 'post' column into a format that the model can understand—specifically, token IDs and attention masks.

1. `input_ids`: The token IDs generated from the tokenization process.
2. `attention_mask`: Indicates which tokens should be attended to (1) and which are padding (0).
3. `label`: The sentiment labels for the classification task.

Model choice

DistilBERT was chosen for this study due to its efficiency, compact size, and strong performance. Developed as a lighter and faster variant of BERT (Bidirectional Encoder Representations from Transformers), DistilBERT retains most of the performance characteristics of BERT. This makes it an ideal choice for natural language processing tasks, where computational efficiency and quick inference times are critical.

This model is based on the Transformer architecture, consisting of an encoder stack. However, unlike BERT, which uses 12 encoder layers in its base version, DistilBERT uses only 6 encoder layers. This reduction in the number of layers is one of the primary reasons for its smaller size and faster processing capabilities.

Each encoder layer in DistilBERT contains 12 attention heads, similar to BERT, allowing the model to focus on different parts of the input sequence simultaneously.

It maintains the hidden size of 768 units per layer, the same as BERT, ensuring that each layer can still capture a substantial amount of information from the input. The feedforward layers within each encoder block have a dimensionality of 3072, contributing to the model's ability to learn and represent intricate patterns in the data.

DistilBERT is designed to be approximately 60% faster than BERT, making it highly suitable for real-time applications. It has 66 million of parameter twice less than the original model but still retains it's 97% of performance.

In conclusion, DistilBERT's structure with its reduced number of parameters and almost the same performance levels, provides a practical and powerful tool in our case.

Here you can see the structure of model described above:
`DistilBertForSequenceClassification`(

`(distilbert): DistilBertModel`(

`(embeddings): Embeddings`(

`(word_embeddings): Embedding`(30522, 768, padding_idx=0)

```

(position_embeddings): Embedding(512, 768)
(LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
(dropout): Dropout(p=0.1, inplace=False)
)
(transformer): Transformer(
(layer): ModuleList(
(0-5): 6 x TransformerBlock(
(attention): MultiHeadSelfAttention(
(dropout): Dropout(p=0.1, inplace=False)
(q_lin): Linear(in_features=768, out_features=768, bias=True)
(k_lin): Linear(in_features=768, out_features=768, bias=True)
(v_lin): Linear(in_features=768, out_features=768, bias=True)
(out_lin): Linear(in_features=768, out_features=768, bias=True)
)
(sa_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
(ffn): FFN(
(dropout): Dropout(p=0.1, inplace=False)
(lin1): Linear(in_features=768, out_features=3072, bias=True)
(lin2): Linear(in_features=3072, out_features=768, bias=True)
(activation): GELUActivation()
)
(output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
)
)
)
)
(pre_classifier): Linear(in_features=768, out_features=768, bias=True)
(classifier): Linear(in_features=768, out_features=3, bias=True)

```

(dropout): Dropout(p=0.2, inplace=False)

So let's move to evaluating of baseline model. To perform this action I froze all layers, stopping updating of weights and biases, and used initial arguments of TrainingArguments in library Transformer.

```
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=3)
model.to(device)

for param in model.parameters():
    param.requires_grad = False
```

```
training_args = TrainingArguments(
    output_dir='content/results ',
    do_train=False,
    do_eval=True,
    logging_dir='content/logs',
    logging_steps=400,
    eval_strategy='epoch',
    save_strategy='epoch',
    fp16=True,
    seed=42,
)

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

After evaluating we got quite poor values of metrics:

1. eval_loss: 1.1015896797180176
2. eval_accuracy: 0.3603385731559855
3. eval_f1: 0.2657722506078441
4. eval_precision: 0.23882212033510417
5. eval_recall: 0.3603385731559855

As preclassifier and classifier layer are initialized by random values, we should train this model on downstream task to better its predictions.

Layer unfreezing

And it leads us to first fine-tuning techniques – layer unfreezing. What does it actually mean? It involves selectively allowing specific layers of a pre-trained model to update their weights and biases during training. When a model is initially pre-trained on a large corpus of general text, its layers learn representations that are broadly useful for a variety of tasks. However, these pre-trained weights might not be entirely optimal for specific tasks, like sentiment analysis on Twitter data, where domain-specific nuances are critical.

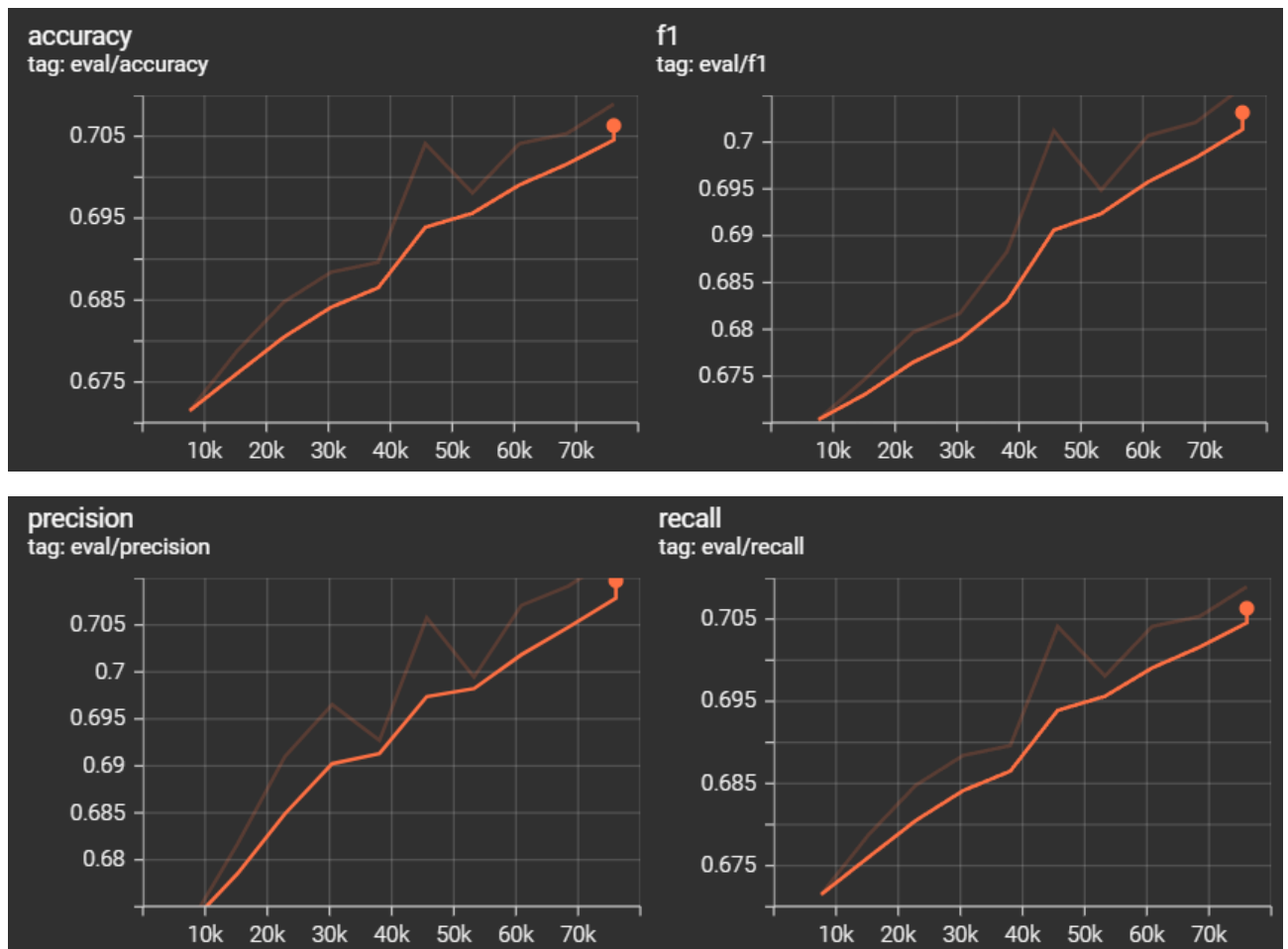
After looking on structure above we unfroze preclassifier and classifier layers to update its parameters for our specific task.

```
parameters_to_unfreeze = ['classifier.bias', 'classifier.weight',  
                           'pre_classifier.bias', 'pre_classifier.weight']  
for name, param in model.named_parameters():  
    if name in parameters_to_unfreeze:  
        param.requires_grad = True
```

After unfreezing specific layers we can train and evaluate model again with default arguments of **TrainingArguments**. And we get such results:

1. eval_loss: 0.6765
2. eval_accuracy: 0.7089
3. eval_f1: 0.7059
4. eval_precision: 0.7097
5. eval_recall: 0.7089

As we can see results are much better, but they are still far from desired ones. The reason of it is lack of data. Updating parameters of two layers requires more data and also it must be more multidisciplinary.



Checking this plots, we can see all metrics (precision, recall, accuracy, and F1 score) demonstrate a consistent improvement as the training progresses. Some

fluctuations suggests that the model's performance varies at certain steps, but the overall trend is positive.

The model improves with training, and it seems to have potential for further optimization to reduce the variability in the metrics. With increase of epoch value we can get a little bit better results, but it is still not perfect.

Hyperparameter search

Next step of fine-tuning this model is hyperparameter search. For this task Optuna library was used.

At first what is the difference between hyperparameters and parameters. Model parameters are learned during training like weights and biases, while hyperparameters are defined by user.

Optuna is an open-source hyperparameter optimization framework designed to automate the search for the best hyperparameter values. It uses a technique called Bayesian optimization, which builds a probabilistic model of the objective function and uses it to select the most promising hyperparameters to evaluate. It requires less steps than grid or random search as it chooses hyperparameter based on previous search.

It requires an objective function that it will optimize. This function typically includes the training and evaluation of the model with a specific set of hyperparameters.

```
def objective(trial):
    num_train_epochs = trial.suggest_int('num_train_epochs', 2, 15)
    per_device_train_batch_size =
trial.suggest_categorical('per_device_train_batch_size', [8, 16, 32])
    learning_rate = trial.suggest_loguniform('learning_rate', 1e-5, 5e-4)
    warmup_steps = trial.suggest_int('warmup_steps', 0, 500)
    weight_decay = trial.suggest_loguniform('weight_decay', 1e-6, 1e-2)
```

A study in Optuna is an optimization run consisting of multiple trials. Each trial corresponds to a single set of hyperparameter values that Optuna evaluates by running the objective function. In each trial, Optuna suggests values for the hyperparameters using a sampler, which could be a random sampler initially and a more sophisticated Bayesian sampler as the study progresses. The objective function is run with the suggested hyperparameters, and the resulting performance metric (e.g., accuracy) is recorded.

Explanation of hyperparameters:

1. num_train_epochs (2 to 15): This is the number of times the entire training dataset will pass through the model.

2. `per_device_train_batch_size` ([8, 16, 32]): This hyperparameter controls the number of samples processed before the model is updated.
3. `learning_rate` (1e-5 to 5e-4): The learning rate determines how much to adjust the model weights in response to the estimated error each time the model weights are updated.
4. `warmup_steps` (0 to 500): Warmup steps gradually increase the learning rate at the start of training.
5. `weight_decay` (1e-6 to 1e-2): Weight decay is a regularization technique that helps prevent overfitting by reducing large weights.

Also it is important to define sample and pruner of Optuna study:

```
study = optuna.create_study(study_name='hyperparam_tuning',
                           sampler=optuna.samplers.TPESampler(seed=42),
                           direction='maximize',

pruner=optuna.pruners.MedianPruner(n_startup_trials=5, n_warmup_steps=5,
interval_steps=1))

study.optimize(objective, n_trials=40)
best_params = study.best_params
```

Explanation of parameters:

1. `sampler`: The TPE (Tree-structured Parzen Estimator) sampler is used for efficient hyperparameter sampling.
2. `direction='maximize'`: Instructs Optuna to maximize the target metric (accuracy in this case).
3. `pruner`: Uses a median pruner to terminate underperforming trials early, saving computational resources. It prunes trials that are unlikely to outperform the current best based on median evaluation results.

Parameters for Pruner:

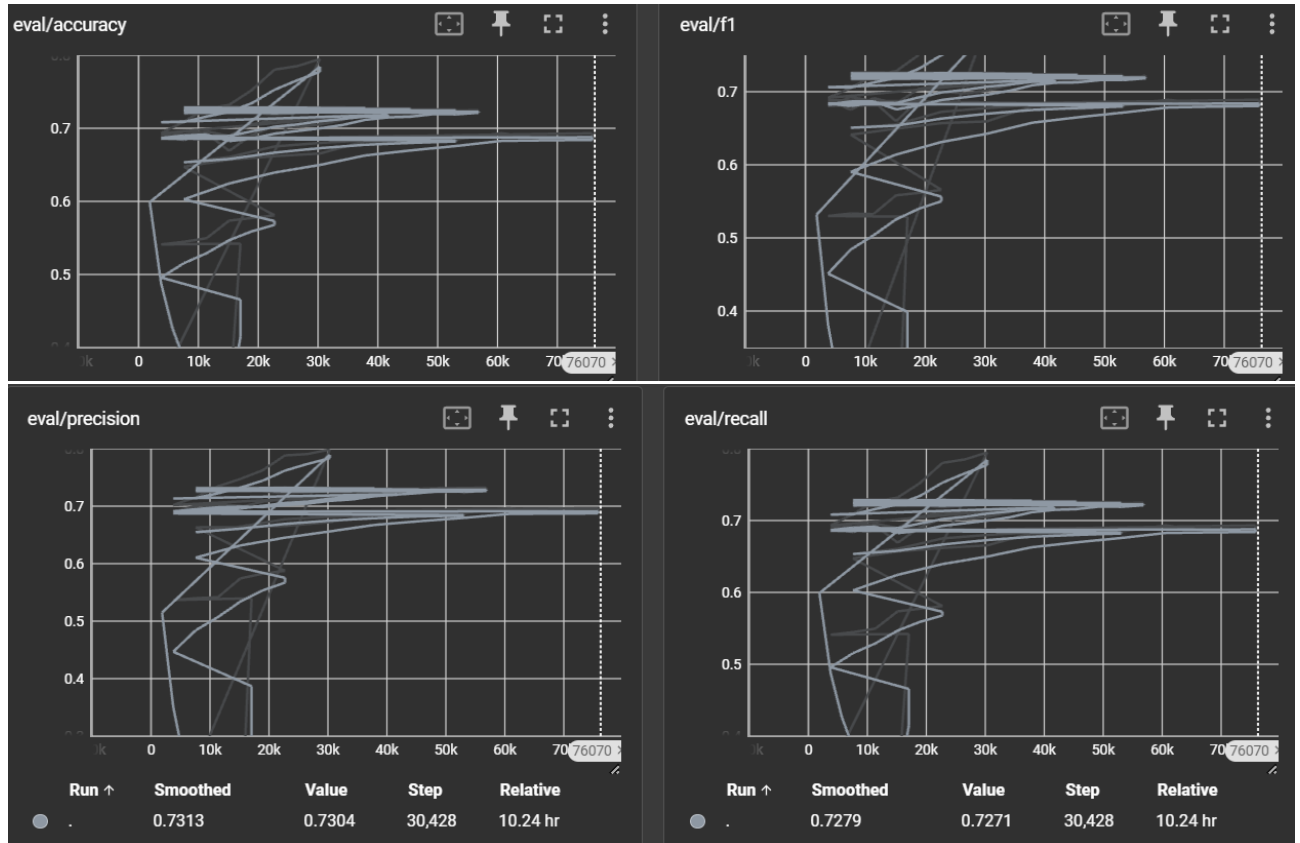
- a. `n_startup_trials=5`: This parameter specifies the number of initial trials that must be completed before the pruner starts.
- b. `n_warmup_steps=5`: This parameter sets the minimum number of evaluation steps (or epochs) that each trial must complete before the pruning.
- c. `interval_steps=1`: This parameter determines the frequency, in terms of steps or epochs, at which the pruner checks whether to prune a trial.

Here we find out the best hyperparameters:

```
best_params = {'num_train_epochs': 8,
               'per_device_train_batch_size': 16,
               'learning_rate': 0.0006672367170464204,
```

```
'warmup_steps': 393,  
'weight_decay': 6.290644294586145e-06}
```

Every trial launches its own training and evaluating with different bunch of hyperparameters, some trials can be pruned or stopped using callback, so its quite hard to analyze plots:



Also it is important to admit that classifier and preclassifier layers were unfrozen for this search.

With the best hyperparameters we get such results:

1. eval_loss: 0.6374
2. eval_accuracy: 0.7295
3. eval_f1: 0.7265
4. eval_precision: 0.7324
5. eval_recall: 0.7295

Results are better in comparison to layer-unfreezing ones. Although we have some improvements but it still far from ideal model.

Prompt tuning

After unfreezing specific layers and finding out the best hyperparameters for this model we move to next step prompt tuning. Prompt tuning is a technique used to improve how a model responds by giving it specific hints or context during training. Instead of just updating model's parameters like traditional fine-tuning does, prompt tuning adjusts the input text by adding prompts before or after the main content.

```
def add_prompt(examples, prompt: str)->dict:
    examples['post'] = [prompt + post for post in examples['post']]
    return examples
```

```
def tokenize_data(tokenizer: DistilBertTokenizerFast, dataset: Dataset, prompt:
str = None) -> Dataset:

    if prompt:
        dataset = dataset.map(lambda examples: add_prompt(examples, prompt),
batched=True)
```

Such techniques demands freezing all layers but the best hyperparameters were used:

```
training_args = TrainingArguments(
    output_dir='results/prompt_tuning',
    num_train_epochs=best_params['num_train_epochs'],
    per_device_train_batch_size=best_params['per_device_train_batch_size'],
    per_device_eval_batch_size=64,
    do_train=True,
    do_eval=True,
    logging_dir='logs/prompt_tuning',
    logging_steps=400,
    learning_rate=best_params['learning_rate'],
    lr_scheduler_type='linear',
    warmup_steps=best_params['warmup_steps'],
    weight_decay=best_params['weight_decay'],
    eval_strategy='epoch',
    save_strategy='epoch',
    fp16=True,
    seed=42,
    load_best_model_at_end=True,
    metric_for_best_model='accuracy'
)
```

For prompt tuning was used PromptTuningConfig from peft library:

```
config = PromptTuningConfig(
    peft_type='PROMPT_TUNING',
    task_type=TaskType.SEQ_CLS,
    num_virtual_tokens=15,
    num_transformer_submodules=1,
    num_attention_heads=model.config.num_attention_heads,
    num_layers=model.config.num_hidden_layers,
    token_dim=model.config.dim,
    prompt_tuning_init='TEXT',
    prompt_tuning_init_text="Predict if sentiment of this review is positive,
negative, or neutral.",
    tokenizer_name_or_path=MODEL_NAME
```

It is configuration class that specifies how prompt tuning should be applied to a model. This class allows you to define various aspects of prompt tuning, such as the type of tuning, task type, and the structure of the prompt embeddings.

Main parameters of PromptTuningConfig:

1. `peft_type`: Specifies the type of adapter for fine-tuning. It can be LORA or PREFIX_TUNING. For prompt tuning, it is set to 'PROMPT_TUNING'.
2. `task_type`: Indicates the type of task the model is being fine-tuned for. `TaskType.SEQ_CLS` is for sequence classification tasks.
3. `num_virtual_tokens`: Defines the number of trainable virtual tokens (prompt tokens) that will be prepended to each input sequence. These tokens are essentially learnable embeddings.
4. `num_transformer_submodules`: Specifies how many submodules (e.g., layers or attention heads) within the model the prompt tokens will interact with. For models like DistilBERT, this is typically set to 1 since the architecture uses a single stack of transformer layers. You can see that in structure of model that was shown above.
5. `num_attention_heads` and `num_layers`: These parameters correspond to the model's architecture, setting the number of attention heads and layers the prompt tokens will interact with.
6. `token_dim`: Sets the dimensionality of the prompt tokens, matching the hidden size of the model.
7. `prompt_tuning_init`: Specifies how the prompt tokens should be initialized. 'TEXT': Initializes the prompt tokens with text-based embeddings derived from a prompt sentence.
8. `prompt_tuning_init_text`: If initializing with text, this specifies the actual text used to create the initial prompt embeddings.
9. `tokenizer_name_or_path`: Path or name of the tokenizer.

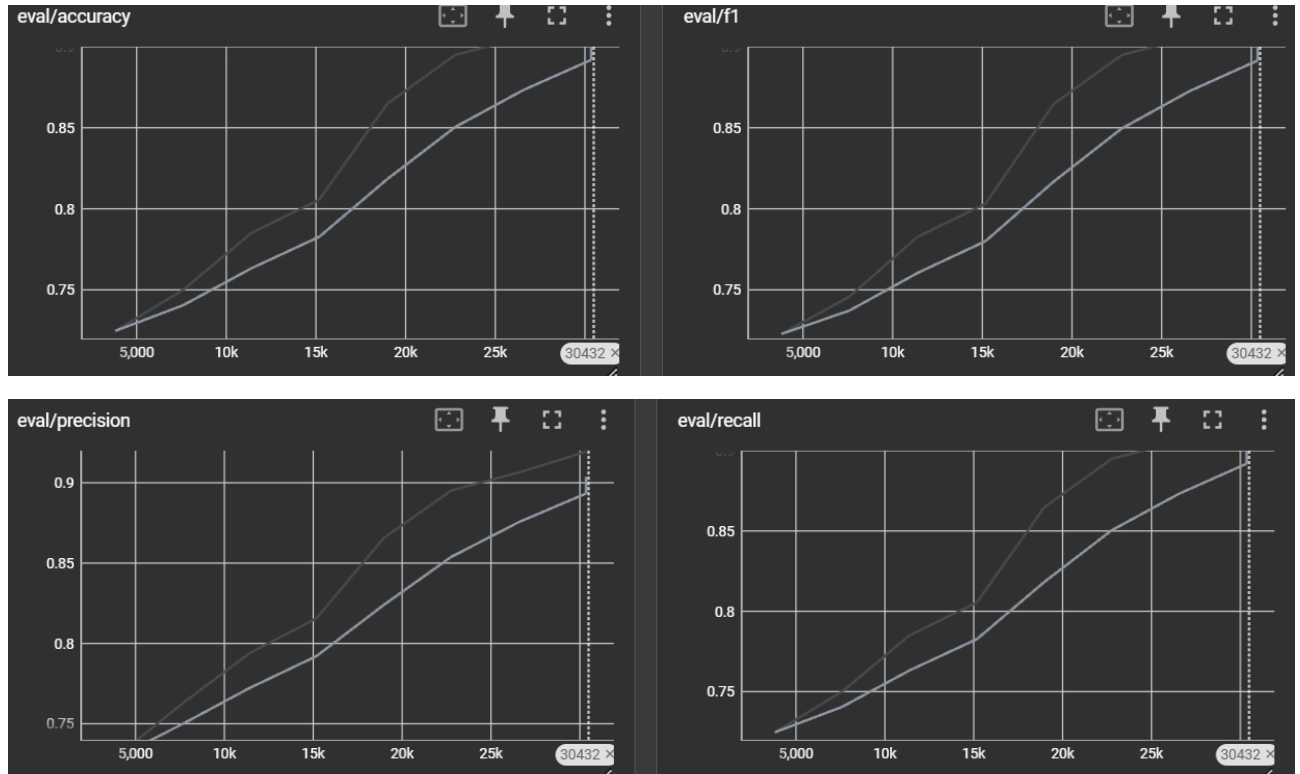
But how will model study if all layers are frozen? Instead of modifying the model's weights, prompt tuning initializes a set of learnable embeddings(soft prompts). These embeddings are prepended to the input tokens. During training, only these prompt embeddings are updated based on the task-specific data, while the rest of the model remains unchanged.

Here we have three prompts that will be evaluated:

```
prompts = [
    "The sentiment of this review is:",
    "This tweet expresses a sentiment that is:",
    "Sentiment classification of this message:",
]
```

After training and evaluating model on every prompt independently we get that third prompt provides the best results:

1. eval_loss: 0.273
2. eval_accuracy: 0.9191
3. eval_f1: 0.91
4. eval_precision: 0.9194
5. eval_recall: 0.9191



All metrics are improving consistently as the number of steps increases, suggesting that the training process is effective. Also if we increase epoch value, we can get even better results.

Conclusion

The study found that while initial preprocessing steps like removing irrelevant tweets and standardizing text improved data quality, advanced techniques such as lemmatization and stopword removal negatively impacted model performance. Layer unfreezing enhanced the baseline model by updating weights in specific layers, but improvements were limited by the dataset's size and diversity, highlighting the need for more data. Hyperparameter optimization with Optuna further improved performance, though results were almost the same compared to layer unfreezing.

Prompt tuning delivered the most significant performance boost, achieving an accuracy of 91.91% by adding contextual prompts like "Sentiment classification of this message" and adjusting only prompt embeddings. This approach effectively balances computational efficiency with performance, making it suitable for real-time applications. Being that most effective and the least resource-demanding technique, it is stated that prompt tuning is the best approach for such case.