Fine-tuning RoBERTa for Wine click prediction with assortment

compiled by D.Gueorguiev, 4/19/2025

Table of Contents

[Roberta Model for Sequence Classification 2](#_Toc196551212)

[Flow of raw input text through the pretrained RoBERTa model 4](#_Toc196551213)

[Steps in the process of fine tuning of the model 5](#_Toc196551214)

[The Multi-Class Classifier Models Considered for the Prediction with Assortment 11](#_Toc196551215)

[Using the WineID as a class identifier 11](#_Toc196551216)

[Using product\_embed\_description as a class identifier 11](#_Toc196551217)

[Using compound query string as input text 12](#_Toc196551218)

[Summarized Results From the Training Phase 13](#_Toc196551219)

[Using the query string as an input text and the WineID as a class identifier 13](#_Toc196551220)

[Using compound query string as an input text and the WineID as a class identifier 13](#_Toc196551221)

[Evaluation of the Fine Tuned Model 14](#_Toc196551222)

[Using the query string as input text and WineID as a class identifier 14](#_Toc196551223)

[Evaluation using the original test dataset 14](#_Toc196551224)

[Manual tests 15](#_Toc196551225)

[Using the compound query string as input text and WineID as a class identifier 16](#_Toc196551226)

[Interpretation of the test accuracy score in the model evaluation 17](#_Toc196551227)

[Using the query string as input text and WineID as a class identifier 17](#_Toc196551228)

[Using compound query string as input text and WineID as a class identifier 18](#_Toc196551229)

[Debugging the forward pass of Roberta model 18](#_Toc196551230)

[Inside the code which computes the loss 18](#_Toc196551231)

[Inside the Transformer Trainer: Debugging the training loop 20](#_Toc196551232)

[References 20](#_Toc196551233)

[Appendix 21](#_Toc196551234)

[Detailed Results from the Training Phase 21](#_Toc196551235)

[Using the original query string as input text and the WineID as a class identifier 21](#_Toc196551236)

[Using compound query string as input text and the WineID as a class identifier 29](#_Toc196551237)

[Inside HF Transformers Roberta Model 34](#_Toc196551238)

[Roberta Embedding 35](#_Toc196551239)

[Roberta Self Attention 36](#_Toc196551240)

[Roberta Sdpa Self Attention 38](#_Toc196551241)

[Roberta Self Output 40](#_Toc196551242)

[Roberta Attention 40](#_Toc196551243)

[Roberta Intermediate 41](#_Toc196551244)

[Roberta Output 41](#_Toc196551245)

[Roberta Layer 41](#_Toc196551246)

[Roberta Encoder 43](#_Toc196551247)

[Roberta Pooler 44](#_Toc196551248)

[Base pretrained Roberta Model 44](#_Toc196551249)

[General Purpose Roberta Model 45](#_Toc196551250)

[Roberta Model for Sequence Classification 50](#_Toc196551251)

[Insider the HF Transformers Trainer module 52](#_Toc196551252)

[The Trainer class 52](#_Toc196551253)

[Trainer::train 74](#_Toc196551254)

[find\_executable\_batch\_size in trainer\_utils.py 76](#_Toc196551255)

[Trainer::\_inner\_training\_loop 77](#_Toc196551256)

[Trainer::training\_step 84](#_Toc196551257)

[Trainer::compute\_loss 85](#_Toc196551258)

[Loss Functions in Torch 86](#_Toc196551259)

[Cross Entropy 86](#_Toc196551260)

[Binary Cross Entropy 88](#_Toc196551261)

[Binary Cross Entropy With Logits 89](#_Toc196551262)

[Smooth L1 Loss 90](#_Toc196551263)

[L1 loss 91](#_Toc196551264)

[MSE Loss 92](#_Toc196551265)

# Roberta Model for Sequence Classification

We are using base pretrained model RoBERTa for sequence classification and we will apply at the multi-class classification problem which we have at hand ; namely wine classification based on supplied query. First, we will fine tune the base pretrained model RoBERTa with a training dataset and later we will validate it and test it with the validation and testing datasets accordingly.

The RoBERTa base model for sequence classification has Embedding component and Encoder component.

The Embedding component contains word embeddings, positional embeddings, and token type embeddings. After the embedding layers there is a normalization layer and dropout layer for regularization purposes. The embedding component size is 768 neurons and vocabulary size of 50,265 tokens. The position embeddings layer can handle input sequences of up to 514 tokens, which are passed when the model is executed against every instance.

The Encoder component has 12 layers each with individual self-attention, intermediate and output components.

GeLU activation function is used in the intermediate layer. Those layers are followed by a normalization layer and dropout layer, again for regularization purposes. Each encoder layer has a hidden size of 768 neurons.

Notice that the intermediate layer has output size of 3072 neurons.

The final layer, RobertaClassificationHead, is a task-specific classification layer designed for sequence classification tasks. It takes the final hidden state of the CLS token and passes it to through a linear layer and a soft max function to produce class probabilities. In this case the model is configured with 2600 unique Wine identifiers. We pass a list of all of those 2600 wine ids as an input to the model. Also we need to supply bi directional label mappings which are id2label and label2id dicts.

wine\_names = catalog['WineName'].unique().tolist()

wine\_ids = catalog['WineID'].unique().tolist()

NUM\_OF\_WINE\_IDS = len(wine\_ids)

NUM\_OF\_WINE\_NAMES = len(wine\_names)

pprint.pp(f"Number of unique labels: {NUM\_OF\_WINE\_NAMES}")

pprint.pp(f"Number of unique wine ids: {len(wine\_ids)}")

pprint.pp(f"Wine names: {wine\_names[1:6]}")

pprint.pp(f"Wine IDs: {wine\_ids[1:6]}")

'Number of unique labels: 1553'

'Number of unique wine ids: 2600'

("Wine names: ['Bandol Rouge', 'Madiran', 'Cuvée Prestige Madiran', 'Palette "

"Rouge', '10 Year Old Tawny Port']")

'Wine IDs: [111791, 112029, 112169, 112740, 112763]'

model = RobertaForSequenceClassification.from\_pretrained(model\_id, output\_hidden\_states=True, num\_labels=NUM\_OF\_WINE\_IDS, id2label=id2wineid, label2id=wineid2id)

model.to(device)

RobertaForSequenceClassification(

(roberta): RobertaModel(

(embeddings): RobertaEmbeddings(

(word\_embeddings): Embedding(50265, 768, padding\_idx=1)

(position\_embeddings): Embedding(514, 768, padding\_idx=1)

(token\_type\_embeddings): Embedding(1, 768)

(LayerNorm): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

(dropout): Dropout(p=0.1, inplace=False)

)

(encoder): RobertaEncoder(

(layer): ModuleList(

(0-11): 12 x RobertaLayer(

(attention): RobertaAttention(

(self): RobertaSdpaSelfAttention(

(query): Linear(in\_features=768, out\_features=768, bias=True)

(key): Linear(in\_features=768, out\_features=768, bias=True)

(value): Linear(in\_features=768, out\_features=768, bias=True)

(dropout): Dropout(p=0.1, inplace=False)

)

(output): RobertaSelfOutput(

(dense): Linear(in\_features=768, out\_features=768, bias=True)

(LayerNorm): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

(dropout): Dropout(p=0.1, inplace=False)

)

)

(intermediate): RobertaIntermediate(

(dense): Linear(in\_features=768, out\_features=3072, bias=True)

(intermediate\_act\_fn): GELUActivation()

)

(output): RobertaOutput(

(dense): Linear(in\_features=3072, out\_features=768, bias=True)

(LayerNorm): LayerNorm((768,), eps=1e-05, elementwise\_affine=True)

(dropout): Dropout(p=0.1, inplace=False)

)

)

)

)

)

(classifier): RobertaClassificationHead(

(dense): Linear(in\_features=768, out\_features=768, bias=True)

(dropout): Dropout(p=0.1, inplace=False)

(out\_proj): Linear(in\_features=768, out\_features=2600, bias=True)

)

)

# Flow of raw input text through the pretrained RoBERTa model

The result at the end stage of the model is a class prediction (probability score) as a result of class-specific fine tuning.

In this type of BERT-based models the CLS token serves as a special token which is prepended to the input sequence. It is designed to be used as a denotation mark of an aggregate representation of the entire input sequence for classification task. There is sequence of steps in wich the CLS token is handled during the fine tuning for specific classification task:

**Step 1: Tokenization**

During the preprocessing of the input queries (text) the tokenizer inserts CLS token at the beginning of every input sequence. For example if the input text is “This is a sample sentence” then tokenizer will append it as “[CLS]This is a sample sentence”.

**Step 2: Embedding**

The tokenized input sequence including the CLS token is passed through RoBERTa model embedding layer which convert the tokens into continuous value word vectors.

**Step 3: Encoder layer and Self-Attention**

The Encoder layers process the embedded input sequences . Recall, the Encoder layer consists of Self-Attention layer and Feed Forward Neural Net. During this process the model learns to capture the semantic and syntactic information presented in the input sequence as well as any relationships between the tokens.

The final hidden state of CLS is captured at the end of the RoBERTa model Encoder layer. There each token has a corresponding hidden state vector. For the CLS token the final hidden state is used as an aggregated representation of the entire input sequence.

**Step 4: Task-specific classification head**

The final hidden state vector which was presented at the of the Encoder layer is passed as an input to the task-specific classification layer. The Linear layer of the classification head maps the 768-dimensional hidden state vector to a vector of 2600 dimensions where each dimension corresponds to a specific class probability score. This is essentially a weight matrix multiplication followed by addition of a bias term. The final step is applying a softmax function to the vector of raw output values into a vector of probability scores. The softmax function application guarantees that the probability scores across all 2600 classes sum up to 1.

**Step 5: Prediction**

The class with the highest probability is chosen as the final prediction for the given input sequence.

During the fine tuning phase the model adjusts incrementally the weights and biases based on the training data and the target labels. This involves updating the parameters (weights and biases) of the base RoBERTa pretrained model and the parameters (weights and biases) of the pretrained classification layer through backpropagation and stochastic gradient descent.

Thus the 5 step fine-tuning process allows the original pre-trained model to adapt to the classification task at hand and improve its performance to the given dataset.

**Question**: in the fine-tuning process are all of the weights of the pretrained model updated?

The answer is yes, all of the weights of the pretrained RoBERTa model are potentially subject to modification. This includes the weights in the embedding layers, the encoder layers, and the classification layer. However, the extent to which the weights of the layer are modified depends on three things :

i ) the learning rate

ii ) the specific task

iii ) the training data

In general fine-tuning a pretrained model like RoBERTa involves updating its weights to better adapt to the target task. When fine-tuning begins the initial weights of the model come from the pretrained base model, which has already learned general language representation from a large scale unsupervised task (for example [masked language modeling](https://huggingface.co/docs/transformers/en/tasks/masked_language_modeling), see [9]).

During the fine tuning the model is exposed to the particular task-specific training data and labels and the weights are updated using back-propagation and some variant of the gradient descent algorithm. Typically the learning rate for fine tuning is set to be smaller than the learning rate used during pretraining. This is because pretrained model already has a good understanding of language and the fine tuning process aims to make small incremental adjustments to the pretrained weights without losing the valuable general language knowledge that the model had already acquired the pretraining phase.

## Steps in the process of fine tuning of the model

**1 ) Split the datasets**

I have experimented with two *kinds* of datasets –

i ) the original set of datasets which contains 12,932 examples for training and 3,233 examples for testing

ii ) a resampled set of datasets which contains 9,699 examples for training, 3,233 examples for validation and 3,233 examples for testing. The resampled datasets are obtained by aggregating the original training and testing datasets (total 16,165 examples), sampling at random the aggregate and splitting it three-way on 9,699 examples for training, 3,233 examples for validation, and 3,233 examples for testing.

click\_train = pd.read\_parquet(TRAIN\_DATASET\_PATH)

click\_test = pd.read\_parquet(TEST\_DATASET\_PATH)

catalog = pd.read\_parquet(CATALOG\_DATASET\_PATH)

click\_all = pd.concat([click\_train, click\_test], ignore\_index=True)

display(click\_all.head())

print(f"Number of rows in the training dataset: {len(click\_train)}")

print(f"Number of rows in the test dataset: {len(click\_test)}")

print(f"Number of rows in the combined dataset: {len(click\_all)}")

# split into 60% training, 20% test and 20% validation

click\_train\_new, click\_validate\_new, click\_test\_new = \

np.split(click\_all.sample(frac=1, random\_state=42),

[int(.6\*len(click\_all)), int(.8\*len(click\_all))])

Number of rows in the training dataset: 12932

Number of rows in the test dataset: 3233

Number of rows in the combined dataset: 16165

Number of rows in the resampled training dataset: 9699

Number of rows in the resampled validation dataset: 3233

Number of rows in the resampled test dataset: 3233

**2 ) Create and apply the tokenizer instance to the training dataset**

model\_id = "roberta-base"

tokenizer = RobertaTokenizerFast.from\_pretrained(model\_id)

train\_texts = list(click\_train\_new['query'])

val\_texts = list(click\_validate\_new['query'])

test\_texts = list(click\_test\_new['query'])

train\_labels = list(click\_train\_new['label'])

val\_labels = list(click\_validate\_new['label'])

test\_labels = list(click\_test\_new['label'])

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

val\_encodings = tokenizer(val\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

**3 ) Define DataLoader**

class DataLoader(Dataset):

"""

Custom Dataset class for handling tokenized text data and corresponding labels.

Inherits from torch.utils.data.Dataset.

"""

def \_\_init\_\_(self, encodings, labels):

"""

Initializes the DataLoader class with encodings and labels.

Args:

encodings (dict): A dictionary containing tokenized input text data

(e.g., 'input\_ids', 'token\_type\_ids', 'attention\_mask').

labels (list): A list of integer labels for the input text data.

"""

self.encodings = encodings

self.labels = labels

def \_\_getitem\_\_(self, idx):

"""

Returns a dictionary containing tokenized data and the corresponding label for a given index.

Args:

idx (int): The index of the data item to retrieve.

Returns:

item (dict): A dictionary containing the tokenized data and the corresponding label.

"""

# Retrieve tokenized data for the given index

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

# Add the label for the given index to the item dictionary

item['labels'] = torch.tensor(self.labels[idx])

return item

def \_\_len\_\_(self):

"""

Returns the number of data items in the dataset.

Returns:

(int): The number of data items in the dataset.

"""

return len(self.labels)

On the line of \_\_getitem\_\_ involving dictionary comprehension

# Retrieve tokenized data for the given index

item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}

This line converts the encoding values associated with the input text (in our case wine query) into a pytorch tensor.

The dictionary self.encodings contains tokenized input text with keys like *input ids*, *token type ids*, and *attention mask*.

These keys represent defined aspects of the encoded text that are needed for the processing by the RoBERTa model. Values associated with these keys are lists or arrays of integers. Thus, this dict comprehension line transforms the values of the input ids, token type ids, and the attention masks into pytorch tensors. We invoke this class on the train, validation and test encodings obtained on the tokenization step to obtain the train , validation, and test dataloader instances as shown below:

train\_dataloader = DataLoader(train\_encodings, train\_labels)

val\_dataloader = DataLoader(val\_encodings, val\_labels)

test\_dataloader = DataLoader(test\_encodings, test\_labels)

**4 ) Define compute\_metrics function**

from sklearn.metrics import accuracy\_score, precision\_recall\_fscore\_support

def compute\_metrics(pred):

"""

Computes accuracy, F1, precision, and recall for a given set of predictions.

Args:

pred (obj): An object containing label\_ids and predictions attributes.

- label\_ids (array-like): A 1D array of true class labels.

- predictions (array-like): A 2D array where each row represents

an observation, and each column represents the probability of

that observation belonging to a certain class.

Returns:

dict: A dictionary containing the following metrics:

- Accuracy (float): The proportion of correctly classified instances.

- F1 (float): The macro F1 score, which is the harmonic mean of precision

and recall. Macro averaging calculates the metric independently for

each class and then takes the average.

- Precision (float): The macro precision, which is the number of true

positives divided by the sum of true positives and false positives.

- Recall (float): The macro recall, which is the number of true positives

divided by the sum of true positives and false negatives.

"""

# Extract true labels from the input object

labels = pred.label\_ids

# Obtain predicted class labels by finding the column index with the maximum probability

preds = pred.predictions.argmax(-1)

# Compute macro precision, recall, and F1 score using sklearn's precision\_recall\_fscore\_support function

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, preds, average='macro')

# Calculate the accuracy score using sklearn's accuracy\_score function

acc = accuracy\_score(labels, preds)

# Return the computed metrics as a dictionary

return {

'Accuracy': acc,

'F1': f1,

'Precision': precision,

'Recall': recall

}

The actual prediction of the most probable class which should be assigned to the new query happens in the line:

# Obtain predicted class labels by finding the column index with the maximum probability

preds = pred.predictions.argmax(-1)

which returns the index of the max value for all class probabilistic scores.

The next line calculates the macro precision, the recall and the F1 score.

Precision is the fraction of true positives among all positives recalled:

Recall is the fraction of the true positives among all events which in reality are positives:

F1 score amalgamates the Precision and the Recall in a single score using the formula:

# Compute macro precision, recall, and F1 score using sklearn's precision\_recall\_fscore\_support function

precision, recall, f1, \_ = precision\_recall\_fscore\_support(labels, preds, average='macro')

The accuracy score implemented in scikit-learn measures the proportion of correctly predicted labels to the total number of labels. In essence, it answers the question: "*Out of all the predictions, what percentage did the model get right?*"

For details see [10].

**5 ) Defining the Training Configuration and creating a model instance**

model\_id = "roberta-base"

repository\_id = "dimitarpg13/roberta-base\_wines"

# Update the model's configuration with the id2label mapping

config = AutoConfig.from\_pretrained(model\_id)

config.update({"id2label": id2wineid})

config.update({"output\_hidden\_states":True})

# Model

model = RobertaForSequenceClassification.from\_pretrained(model\_id, config=config)

# TrainingArguments

training\_args = TrainingArguments(

output\_dir=repository\_id,

num\_train\_epochs=5,

per\_device\_train\_batch\_size=8,

per\_device\_eval\_batch\_size=8,

#evaluation\_strategy="epoch",

eval\_strategy="epoch",

logging\_dir=f"{repository\_id}/logs",

logging\_strategy="steps",

logging\_steps=10,

learning\_rate=5e-5,

weight\_decay=0.01,

warmup\_steps=500,

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

save\_total\_limit=2,

report\_to="tensorboard",

push\_to\_hub=True,

hub\_strategy="every\_save",

hub\_model\_id=repository\_id,

hub\_token=HfFolder.get\_token(),

)

The number or epochs, the weight decay, the learning rate, per\_device\_train\_batch\_size and warmup\_steps are subject of experimentation. The more epochs we are using the higher the accuracy score should be.

**5 ) Training with the Trainer class**

Usually we would instantiate the Trainer class with appropriate arguments and then we call its method train().

However, because we have a custom loss function we will subclass the Trainer class and override the method compute\_loss(..) supplying our own custom loss function instead. For some additional details on this procedure see [11],[12],[13],and [14].

# Define a custom Trainer with the loss function

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, num\_items\_in\_batch, return\_outputs=False):

labels = inputs.get("labels")

# forward pass

outputs = model(\*\*inputs)

logits = outputs.get("logits") # the output embeddings

# compute custom loss

loss\_fct = torch.nn.CrossEntropyLoss(weight=None)

loss = loss\_fct(logits.view(-1, self.model.config.num\_labels), labels.view(-1))

return (loss, outputs) if return\_outputs else loss

# Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataloader,

eval\_dataset=val\_dataloader,

)

# start the training process.

trainer.train()

**6 ) Validate the fine-tuned model**

def predict(text):

"""

Predicts the class label for a given input text

Args:

text (str): The input text for which the class label needs to be predicted.

Returns:

probs (torch.Tensor): Class probabilities for the input text.

pred\_label\_idx (torch.Tensor): The index of the predicted class label.

pred\_label (str): The predicted class label.

"""

# Tokenize the input text and move tensors to the GPU if available

inputs = tokenizer(text, padding=True, truncation=True, max\_length=514, return\_tensors="pt").to("cuda")

# Get model output (logits)

outputs = model(\*\*inputs)

probs = outputs[0].softmax(1)

""" Explanation outputs: The RoBERTa model returns a tuple containing the output logits (and possibly other elements depending on the model configuration). In this case, the output logits are the first element in the tuple, which is why we access it using outputs[0].

outputs[0]: This is a tensor containing the raw output logits for each class. The shape of the tensor is (batch\_size, num\_classes) where batch\_size is the number of input samples (in this case, 1, as we are predicting for a single input text) and num\_classes is the number of target classes.

softmax(1): The softmax function is applied along dimension 1 (the class dimension) to convert the raw logits into class probabilities. Softmax normalizes the logits so that they sum to 1, making them interpretable as probabilities. """

# Get the index of the class with the highest probability

# argmax() finds the index of the maximum value in the tensor along a specified dimension.

# By default, if no dimension is specified, it returns the index of the maximum value in the flattened tensor.

pred\_label\_idx = probs.argmax()

# Now map the predicted class index to the actual class label

# Since pred\_label\_idx is a tensor containing a single value (the predicted class index),

# the .item() method is used to extract the value as a scalar

pred\_label = model.config.id2label[pred\_label\_idx.item()]

return probs, pred\_label\_idx, pred\_label

# Test with an example query

text = "cabernet sauvignon shiraz"

predict(text)

# The Multi-Class Classifier Models Considered for the Prediction with Assortment

## Using the WineID as a class identifier

# add wine id of the clicked wine to the training , test, and validation data

def add\_label(row):

indices = np.where(row.new\_labels == 1)

wine\_id = int(row.results[indices[0][0]])

row['label'] = wineid2id[wine\_id]

return row

click\_train = click\_train.apply(lambda row: add\_label(row), axis=1)

click\_test = click\_test.apply(lambda row: add\_label(row), axis=1)

click\_train\_new = click\_train\_new.apply(lambda row: add\_label(row), axis=1)

click\_test\_new = click\_test\_new.apply(lambda row: add\_label(row), axis=1)

click\_validate\_new = click\_validate\_new.apply(lambda row: add\_label(row), axis=1)

## Using product\_embed\_description as a class identifier

I was not able to get past 10% Accuracy rate with the WineID as class id. So I decided to try using product\_embed\_description as a class identifier. I thought product\_embed\_description could have different semantics than Wine ID. But to my surprise I found that there is a unique correspondence (bijection) between product\_embed\_description strings and the Wine IDs in the catalog. That is – using product\_embed\_description is equivalent to using WineID.

df = catalog[['WineName',"WineID", "product\_embed\_description"]]

def update(row, wineid2winename, wineid2ped):

wineid2winename.update({row.WineID: row.WineName})

wineid2ped.update({row.WineID: row.product\_embed\_description})

# create wineid2winename and wineid2ped dict

wineid2winename = dict()

wineid2ped = dict()

df.apply(lambda row: update(row, wineid2winename, wineid2ped), axis=1)

pprint.pp(f"WineID to WineName dict (5 items): {take(5, wineid2winename.items())}")

pprint.pp(f"Number of unique WineIDs in the catalog: {len(set(list(wineid2ped.keys())))}")

pprint.pp(f"Number of unique product\_embed\_descriptions in the catalog: {len(set(list(wineid2ped.values())))}")

pprint.pp(f"Number of unique wine names in the catalog: {len(set(list(wineid2winename.values())))}")

'Number of unique WineIDs in the catalog: **2600**'

'Number of unique product\_embed\_descriptions in the catalog: **2600**'

'Number of unique wine names in the catalog: 1553'

From the last output stating that there are only 1553 unique wine names , it is obvious that there is many-to-1 correspondence between wine ids and the wine names.

## Using compound query string as input text

Instead of the original query string the new compound query will be a concatenation of the original query string and the input converted to string. In order to limit the number of tokens generated by the tokenizer we limit the number of appended wine ids from the results field to 8.

Example compound query strings:

'italian red wine pairings for game meat 136065 136581 136200 137612 135948 137112 135833 135834',

'vinho de portugal red wine 101576 155465 102122 102127 101682',

'white wines from venica & venica winery 174403 137037 193520 137392 101886',

'$50-$75 red wine for special occasion 174211 112007 179528 112116 111614',

'yalumba cabernet sauvignon shiraz 174279 174186 180111 174195 111413 174295',

'$40-$60 zinfandel 179370 179216 179026 155479 111577 112157',

'very full-bodied red wines under $200 135922 156043 179332 137381',

'best malbec for beef and lamb 167808 162819 167429 167622 167470 155510',

'line 39 pinot noir 112025 193706 179235',

'$50-$60 pinot noir recommendations 180132 179213 167470 179534 179378 193725',

'pinot noir for a special occasion 111488 111813 179561 179642 179965',

'south african chardonnay from hemel-en-aarde valley 180069 170995 171013',

'best german riesling under $100 106561 106610 106606 106591',

'pinot noir aged for a long time 112337 100070 111487'

# create compound query in the training , test and validation data

def add\_compound\_query(row):

result\_set = set(row.results)

if len(result\_set) > 8:

result\_set = list(result\_set)[:8]

result\_set = set(result\_set)

row['compound\_query'] = row.query + " " + str([int(v) for v in result\_set])[1:-1].replace(",", "")

return row

click\_train = click\_train.apply(lambda row: add\_compound\_query(row), axis=1)

click\_test = click\_test.apply(lambda row: add\_compound\_query(row), axis=1)

click\_train\_new = click\_train\_new.apply(lambda row: add\_compound\_query(row), axis=1)

click\_test\_new = click\_test\_new.apply(lambda row: add\_compound\_query(row), axis=1)

click\_validate\_new = click\_validate\_new.apply(lambda row: add\_compound\_query(row), axis=1)

Then we use the newly added compound\_query field to the training and test dataframes as an input text which is passed to tokenizer:

train\_texts = list(click\_train['compound\_query'])

test\_texts = list(click\_test['compound\_query'])

train\_labels = list(click\_train['label'])

test\_labels = list(click\_test['label'])

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

# Summarized Results From the Training Phase

## Using the query string as an input text and the WineID as a class identifier

# TrainingArguments

training\_args = TrainingArguments(

output\_dir=repository\_id,

num\_train\_epochs=200,

per\_device\_train\_batch\_size=32,

per\_device\_eval\_batch\_size=32,

#evaluation\_strategy="epoch",

eval\_strategy="epoch",

logging\_dir=f"{repository\_id}/logs",

logging\_strategy="epoch",

logging\_steps=1,

learning\_rate=1e-5,

weight\_decay=0.01,

warmup\_steps=5,

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

save\_total\_limit=2,

fp16=True,

report\_to="tensorboard",

push\_to\_hub=True,

hub\_strategy="every\_save",

hub\_model\_id=repository\_id,

hub\_token=HfFolder.get\_token(),

)

Cross entropy loss function has been used to train the model.

**[81000/81000 1:39:57, Epoch 200/200]**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Epoch | Training Loss | Validation Loss | Accuracy | F1 | Precision | Recall |
| 200 | 0.428300 | 1.420119 | **0.789669** | 0.709872 | 0.834004 | 0.795186 |

TrainOutput(global\_step=81000, training\_loss=2.2710581553894795, metrics={'train\_runtime': 5997.4878, 'train\_samples\_per\_second': 431.247, 'train\_steps\_per\_second': 13.506, 'total\_flos': 4.62442691165952e+16, 'train\_loss': 2.2710581553894795, 'epoch': 200.0})

Notice that the whole training process contains 81,000 iterations. Each epoch contains 81,000 / 200 = 405 iterations.

The batch size was configured to be 32 instances.

81,000 iterations / 32 = 12,960 examples which is exactly the size of the training dataset.

Also notice that the final test accuracy stated to be **0.79** in the training phase is misleading. The reason the accuracy is so high is because I was using as a validation set a set constructed by resampling the aggregated set of original training and test data. This means that the validation set contains examples which were already used in the training process. As soon as the resampled validation set is replaced with the original test set the final test accuracy drops to 10%.

## Using compound query string as an input text and the WineID as a class identifier

Instead of the original query string the new compound query will be a concatenation of the original query string and the input converted to string. In order to limit the number of tokens generated by the tokenizer we limit the number of appended wine ids from the results field to 8.

# TrainingArguments

training\_args = TrainingArguments(

output\_dir=repository\_id,

num\_train\_epochs=150,

per\_device\_train\_batch\_size=32,

per\_device\_eval\_batch\_size=32,

eval\_strategy="epoch",

logging\_dir=f"{repository\_id}/logs",

logging\_strategy="epoch",

logging\_steps=1,

learning\_rate=1e-5,

weight\_decay=0.01,

warmup\_steps=5,

save\_strategy="epoch",

load\_best\_model\_at\_end=True,

save\_total\_limit=2,

fp16=True,

report\_to="tensorboard",

push\_to\_hub=True,

hub\_strategy="every\_save",

hub\_model\_id=repository\_id,

hub\_token=HfFolder.get\_token(),

)

Cross entropy loss function has been used to train the model.

**[60750/60750 1:21:19, Epoch 150/150]**

Epoch Training Loss Validation Loss Accuracy F1 Precision Recall

150 0.803700 5.215966 0.130529 0.085578 0.481202 0.264804

TrainOutput(global\_step=60750, training\_loss=2.825293725284529, metrics={'train\_runtime': 4879.8661, 'train\_samples\_per\_second': 397.511, 'train\_steps\_per\_second': 12.449, 'total\_flos': 5.20248027561696e+16, 'train\_loss': 2.825293725284529, 'epoch': 150.0})

Improvement in accuracy with compound query string has been observed compared to using the original query string as input text. Now the accuracy with the test data is measured to be **13%** which is higher than the max accuracy of **10%** observed with the original query string as input text. This accuracy was achieved with less training epochs – 150.

# Evaluation of the Fine Tuned Model

## Using the query string as input text and WineID as a class identifier

### Evaluation using the original test dataset

q=[trainer.evaluate(eval\_dataset=df\_org) for df\_org in [train\_dataloader, val\_dataloader, test\_dataset]]

pd.DataFrame(q, index=["train","val","test"]).iloc[:,:5]

**[405/405 01:55]**

| **eval\_loss** | **eval\_Accuracy** | **eval\_F1** | **eval\_Precision** | **eval\_Recall** |
| --- | --- | --- | --- | --- |
| **train** | 0.285065 | 0.953294 | 0.949877 | 0.975734 | 0.944518 |
| **val** | 1.470713 | 0.783174 | 0.700300 | 0.832866 | 0.788176 |
| **test** | 5.990768 | 0.104238 | 0.073507 | 0.406279 | 0.269333 |

Clearly the accuracy with examples the model has not seen before, that is, the original test dataset, is no more than 10%. I cannot increase the accuracy beyond 10% with various values of the test configuration parameters (TrainingArguments).

I experimented with different number of epochs, learning rates and batch sizes and 10% accuracy is the best I can get.

### Manual tests

#### Example query 1

# Test with an example query

text = "cabernet sauvignon shiraz"

res=predict(text)

res[0].argmax(-1)

tensor([463], device='cuda:0')

id2wineid[463]

174246

catalog[catalog['WineID'] == 174246]

WineName: “Shiraz-Cabernet”

Type: Red

Elaborate: Varietal/100%

Grapes: Cabernet Sauvignon

Harmonize: [Beef, Lamb, Poultry]

ABV: 14.0

Body: Very Full Bodied

Acidity: High

Code: AU

Size: 750ml Wine

Price: USD 5.99

Product\_Embed\_Description:

('Wine Name: Shiraz-Cabernet; Wine Type: Red; Wine Elaborate: Varietal/100%; '

'Grape Sources: Cabernet Sauvignon; Wine Harmonize: Beef, Lamb, Poultry; '

'Alcohol By Volume: 14; Wine Body: Very full-bodied; Wine Acidity: High; '

'Bottle Size: 750ml Wine; Price Currency: USD; Price: 5.99; Country of '

'Origin: Australia; Region of Origin: South Eastern Australia; Winery Name: '

"Rawson's Retreat; Vintage Options: 1960-2021; Average Review Score: 3.444; "

'Professional Review: Big discount on this cab, named after Christopher '

"Rawson Penfold's original stone cottage, which still stands among the vines "

'at Magill Estate, in Adelaide, South Australia. The wines are affordable, '

'everyday, easy-drinking wine and is a classic Australian marriage of fruit '

'power and structure. The Cabernet Sauvignon offers opulent dark berry '

'flavous with a moderately firm structure and rich finish. " From one of '

'Australia\'s legendary wineries, the Penfolds "Rawson\'s Retreat" is made in '

'a style to offer good varietal distrinction for current consumption at a '

'user-friendly price. At $5.99, this is one great deal that is perfect for '

"week day drinking and any party or wedding you're planning this year. With "

'only selected parcels receiving moderate oak influence, the wine is bright, '

'fresh, and fruit forward. The nose is lifted with notes of red currant, '

'cherry, earth, chocolate, light olive, and a hint of mint. On the palate '

'there are soft tannins with very nice balance and a lingering finish. Have '

'this bargain b')

#### Example query 2

# Test with an example query

text = "$5-$10 italian red wine"

res=predict(text)

res[0].argmax(-1)

tensor([463], device='cuda:0')

id2wineid[463]

174246

catalog[catalog['WineID'] == 174246]

WineName: “Shiraz-Cabernet”

Type: Red

Elaborate: Varietal/100%

Grapes: Cabernet Sauvignon

Harmonize: [Beef, Lamb, Poultry]

ABV: 14.0

Body: Very Full Bodied

Acidity: High

Code: AU

Size: 750ml Wine

Price: USD 5.99

#### Example query 3

# Test with an example query

text = "very full bodied red wines"

res=predict(text)

res[0].argmax(-1)

tensor([848], device='cuda:0')

id2wineid[848]

174181

catalog[catalog['WineID'] == 174181]

WineName: “The Boxer Shiraz”

Type: Red

Elaborate: Varietal/100%

Grapes: Cabernet Sauvignon

Harmonize: [Beef, Lamb, Game Meat, Poultry]

ABV: 16.0

Body: Very Full Bodied

Acidity: High

Code: AU

Size: 750ml Wine

Price: USD 159.99

Product Embed Description:

('Wine Name: The Boxer Shiraz; Wine Type: Red; Wine Elaborate: Varietal/100%; '

'Grape Sources: Syrah/Shiraz; Wine Harmonize: Beef, Lamb, Game Meat, Poultry; '

'Alcohol By Volume: 16; Wine Body: Very full-bodied; Wine Acidity: High; '

'Bottle Size: 750ml Wine; Price Currency: USD; Price: 159.99; Country of '

'Origin: Australia; Region of Origin: McLaren Vale; Winery Name: Mollydooker; '

'Vintage Options: 1950-2021; Average Review Score: 4.179; Professional '

'Review: Very deep purple-black in color, the 2011 Velvet Glove is chock full '

'of ripe and spicy blackberries, fresh blueberries and creme de cassis aromas '

'that are accented by Chinese five spice notes and hints of chocolate, '

'vanilla and some tar. Full-bodied, it is ripe and rich with a nice line of '

'acid and medium-firm, fine-grained tannins. Concentrated and persistent on '

'the finish, it shows beautiful elegance and freshness and is a nicely '

'delineated and expressive example of the varietal and its regional home. '

'(LPB)')

## Using the compound query string as input text and WineID as a class identifier

I do not have evaluation results for the case of when the input text is composed by the original query string appended with relevant wine ids from the results field of the dataframe. The reason why I do not have evaluation results is because the A100 GPU ran out of memory after successfully completing the training phase.

**Note:** One has to be mindful about the length of the compound query string because the A100 GPU can run very quickly out of memory with large enough query strings. With large compound query strings it could be worth exploring DistilBERT model which is much smaller model instead of RoBERTa and will be able to fit in the A100 GPU memory with ease.

# Interpretation of the test accuracy score in the model evaluation

## Using the query string as input text and WineID as a class identifier

The accuracy score for the unseen by the model test dataset is evaluated with this model at **0.1**. The question is: why is the accuracy score for predicting the clicked wine id based on the query string so low? Can we do a better job at predicting the clicked wine id?

I just noticed something which I should have thought about earlier: no predictive model (LLM-based or not) can achieve better accuracy in this problem than 1 / average\_number\_of\_wine\_ids\_per\_query .

Here are the details which motivate this claim.

The average number of wine ids per query (shown in the column results) in the test dataset is about 8.

Here is the precise data:

Average number of wine ids per query in the training dataset: 7.877281163006495

Standard Deviation of the number of wine ids per query in the training dataset: 7.302453366148524

Variance of the number of wine ids per query in the training dataset: 53.3258251647739

Average number of wine ids per query in the test dataset: 7.780080420661924

Standard Deviation of the number of wine ids per query in the test dataset: 7.207382362087674

Variance of the number of wine ids per query in the test dataset: 51.94636051333249

My conjecture is that as soon as the user sees the result of his query he clicks on a wine id at random.

I want to expound on the meaning of at random in this context: "the user clicks on a wine id at random" means that the query string itself does not contain any evidence which somehow could be used to establish a causal relation between the value of the query string and the specific choice of the wine id from the results list once the result list of wine ids is known. That is, with this conjecture in place, no predictive model (LLM-based or not) can achieve better accuracy in this problem than 1 / average\_number\_of\_wine\_ids\_per\_query . With average\_number\_of\_wine\_ids\_per\_query equal to **8** the best possible accuracy score of our LLM-based predictive model is **0.125** which is far smaller than the number **0.4** which I am expected to achieve per your document.

The reason why I am seeing accuracy score of **0.1** instead of **0.125** for the unseen test dataset is because the LLM model does not perfect job at predicting the set of wine ids, stored in the column results, resulting from executing specific query string. That is, the probability of correctly guessing the clicked wine id , given a query string is:

Here assuming the conjecture holds I have established that can at most **0.125**. From the accuracy score of the test dataset I am seeing that is **0.1**. This means that can be at most **0.8**. That means that the LLM-based model cannot guess the contents of the results column given a query string with accuracy higher than **0.8** for a test dataset.

## Using compound query string as input text and WineID as a class identifier

Recall, in this case instead of the original query string the new compound query will be a concatenation of the original query string and the input converted to string. In order to limit the number of tokens generated by the tokenizer we limit the number of appended wine ids from the results field to 8.

In this case we see improved accuracy on the unseen test data (inference phase) with the newly finetuned model.

With 150 epochs we observe test data accuracy of **13%** compared to **10%** test accuracy when using the original query string.

Wisely engineered compound query could bring the accuracy even higher. For example, more complex compound query can be expressed as:

**"original\_query\_str feature\_1\_of\_wine\_id\_1 feature\_2\_of\_wine\_id\_1 feature\_3\_of\_wine\_id\_1 ... feature\_1\_of\_wine\_id\_8 feature\_2\_of\_wine\_id\_8 feature\_3\_of\_wine\_id\_3"**

The question really is which features of each wine ID in the results list are the most relevant and should be included in the compound query.

**Note:** One has to be mindful about the length of the compound query string because the A100 GPU can run very quickly out of memory with large enough query strings. With large compound query strings it could be worth exploring DistilBERT model which is much smaller model instead of RoBERTa and will be able to fit in the A100 GPU memory with ease.

The interpretation of the improved accuracy with unseen data in this case is that training the model directly to associate the specified in the query wine ids with the target result improves the predictability of the clicked wine id. The reason why is we are making the query string more informative and the model will need to infer less which results in improved prediction.

# Debugging the forward pass of Roberta model

## Inside the code which computes the loss

def compute\_loss(self, model, inputs, num\_items\_in\_batch, return\_outputs=False):

labels = inputs.get("labels")

# forward pass

outputs = model(\*\*inputs)

logits = outputs.get("logits") # the output embeddings

# compute custom loss with given weight tensor (suppose one has 3 labels with different weights)

# loss\_fct = torch.nn.CrossEntropyLoss(weight=torch.tensor([1.0, 2.0, 3.0]))

loss\_fct = torch.nn.CrossEntropyLoss()

loss = loss\_fct(logits.view(-1, self.model.config.num\_labels), labels.view(-1))

%debug

return (loss, outputs) if return\_outputs else loss

Call Location:

File "/usr/lib/python3.11/bdb.py", line 336, in set\_trace

sys.settrace(self.trace\_dispatch)

> <ipython-input-37-7f3ece9239a7>(87)compute\_loss()

85 loss = loss\_fct(logits.view(-1, self.model.config.num\_labels), labels.view(-1))

86 from IPython.core.debugger import Pdb; Pdb().set\_trace()

---> 87 return (loss, outputs) if return\_outputs else loss

88

89

ipdb> logits.shape

torch.Size([8, 2600])

ipdb> labels.shape

torch.Size([8])

ipdb> self.model.config.num\_labels

2600

ipdb> logits.view(-1, self.model.config.num\_labels)

tensor([[-0.0227, 0.0547, -0.0751, ..., -0.1611, -0.1035, 0.1487],

[-0.0358, 0.0252, 0.0059, ..., -0.1143, -0.1237, 0.1959],

[ 0.0182, 0.0065, -0.0304, ..., -0.1654, -0.1297, 0.1537],

...,

[ 0.0874, 0.0334, -0.0640, ..., -0.0568, -0.1053, 0.1636],

[ 0.0367, -0.0600, 0.0031, ..., -0.1769, -0.1242, 0.2190],

[ 0.0238, 0.0314, 0.0002, ..., -0.1536, -0.1068, 0.1430]],

grad\_fn=<ViewBackward0>)

ipdb> type(logits)

<class 'torch.Tensor'>

ipdb> loss.shape

torch.Size([])

ipdb> loss

tensor(7.8714, grad\_fn=<NllLossBackward0>)

ipdb> type(loss)

<class 'torch.Tensor'>

The logits tensor represents the prediction scores of the language modeling head – these are the scores for each vocabulary token before applying SoftMax function to it. It has a shape of 8 x 2600 because we have defined per\_device\_train\_batch\_size=8 in the TrainingArguments instance and because we have 2600 classes.

In case of unbalanced training set we can supply additional weight tensor as an argument to the custom loss function which uses the CrossEntropy implementation in Torch (see *Loss Functions in Torch* in the **Appendix** for details on the implementation).

With the current training dataset we have the following statistics for the wine ids:

('Average number of clicks for a wine id in the training dataset: '

'5.085332284703107')

('Standard Deviation of the number of clicks for a wine id in the training dataset '

'dataset: 2.935745334066074')

('Variance of the number of clicks for a wine id in the training dataset: '

'8.618600666490726')

'Average number of clicks for a wine id in the test dataset: 1.784216335540839'

('Standard Deviation of the number of clicks for a wine id in the test '

'dataset: 1.0687645393811398')

('Variance of the number of clicks for a wine id in the test dataset: '

'1.1422576406385798')

('Average number of clicks for a wine id in the newly resampled training '

'dataset: 3.881152460984394')

('Standard Deviation of the number of clicks for a wine id in the newly '

'resampled training dataset: 2.365323808221432')

('Variance of the number of clicks for a wine id in the newly resampled '

'training dataset: 5.594756717739137')

('Average number of clicks for a wine id in the newly resampled test dataset: '

'1.8275862068965518')

('Standard Deviation of the number of clicks for a wine id in the newly '

'resampled test dataset: 1.0583683423720849')

('Variance of the number of clicks for a wine id in the newly resampled test '

'dataset: 1.1201435481354345')

The dataset clearly is somewhat imbalanced because both for the training and the test dataset.

So it may make sense to assign lower weights to the classes which are less frequently represented in the distribution.

**Stack trace:**

ipdb> bt

[... skipping 21 hidden frame(s)]

<ipython-input-38-a8483dcbf8c4>(2)<cell line: 0>()

**1** # fine tune the model

----> 2 trainer.train()

/usr/local/lib/python3.11/dist-packages/transformers/trainer.py(2236)train()

**2234**  # Disable progress bars when uploading models during checkpoints to avoid polluting stdout

**2235**  hf\_hub\_utils.disable\_progress\_bars()

-> 2236 return inner\_training\_loop(

**2237**  args=args,

**2238**  resume\_from\_checkpoint=resume\_from\_checkpoint,

/usr/local/lib/python3.11/dist-packages/transformers/trainer.py(2560)\_inner\_training\_loop()

**2558**  )

**2559**  with context():

-> 2560 tr\_loss\_step = self.training\_step(model, inputs, num\_items\_in\_batch)

**2561**

**2562**  if (

/usr/local/lib/python3.11/dist-packages/transformers/trainer.py(3736)training\_step()

**3734**

**3735**  with self.compute\_loss\_context\_manager():

-> 3736 loss = self.compute\_loss(model, inputs, num\_items\_in\_batch=num\_items\_in\_batch)

**3737**

**3738**  del inputs

> <ipython-input-37-7f3ece9239a7>(87)compute\_loss()

85 loss = loss\_fct(logits.view(-1, self.model.config.num\_labels), labels.view(-1))

86 **%debug**

---> 87 return (loss, outputs) if return\_outputs else loss

88

89

As one can see from the last frame on the call stack our custom compute\_loss function override is being executed instead of the base class Trainer::compute\_loss(..). For the code of Trainer::compute\_loss(..) see the **Appendix**.

# Inside the Transformer Trainer: Debugging the training loop

//TODO: add some debugging details for the training loop

# References

[1] [BERT Explained: state of the art language model for NLP, Medium, R. Horev, 2018](https://medium.com/data-science/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270)

[2] [Mastering BERT: A Comprehensive Guide from Beginner to Advanced in NLP, Medium, R. Shaikh, 2023](https://medium.com/@shaikhrayyan123/a-comprehensive-guide-to-understanding-bert-from-beginners-to-advanced-2379699e2b51)

[3] [Pretrained RoBERTa model on HuggingFace: Overview](https://huggingface.co/docs/transformers/en/model_doc/roberta)

[4] [pretrained RoBERTa model for Sequence Classification: RobertForSequenceClassification on HuggingFace](https://huggingface.co/docs/transformers/en/model_doc/roberta#transformers.RobertaForSequenceClassification)

[5] [Note on Bi-encoder and Cross-encoder: which one to use, Medium, S. Mudadla, 2023](https://medium.com/@sujathamudadla1213/bi-encoder-vs-cross-encoder-when-to-use-which-one-4a20edbe6d37)

[6] [Cross-Entropy Loss implementation in PyTorch on pytorch.org/docs](https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html)

[7] [Understanding the CLS token in BERT: Comprehensive Guide, Medium, A. Raj, 2024](https://aditya007.medium.com/understanding-the-cls-token-in-bert-a-comprehensive-guide-a62b3b94a941)

[8] [Why Bert transformer uses [CLS] token for classification instead of average over all tokens? StackOverflow, 2021](https://stackoverflow.com/questions/62705268/why-bert-transformer-uses-cls-token-for-classification-instead-of-average-over)

[9] [Masked Language Modeling on HuggingFace.co/docs](https://huggingface.co/docs/transformers/en/tasks/masked_language_modeling)

[10] <https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html>

[11] [How to use custom loss with Hugging Face, Medium, A. Tovar, Medium, 2023](https://medium.com/deeplearningmadeeasy/how-to-use-a-custom-loss-with-hugging-face-fc9a1f91b39b)

[12] [Finetuning BART using custom loss, HuggingFace discussion, 2021](https://discuss.huggingface.co/t/finetuning-bart-using-custom-loss/4060/4)

[13] [Specify Loss for Trainer / TrainingArguments , HuggingFace discussion, 2021](https://discuss.huggingface.co/t/specify-loss-for-trainer-trainingarguments/10481/2)

[14] [How to specify the loss function when finetuning a model using the HuggingFace TFTrainer Class? StackOverflow, 2021](https://stackoverflow.com/questions/66302371/how-to-specify-the-loss-function-when-finetuning-a-model-using-the-huggingface-t)

[15] [NLLLoss implementation in PyTorch on pytorch.org/docs](https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html)

[16] [Python Loss Functions: The Ultimate Guide, A. Opidi et al, online blog at Neptune.ai, 2025](https://neptune.ai/blog/pytorch-loss-functions)

[17] [Transformer Model outputs on HugginFace.com/docs, latest version](https://huggingface.co/docs/transformers/main_classes/output)

[18] [Where to pick up embedding data from BERT model? HuggingFace discussion, 2022](https://discuss.huggingface.co/t/where-to-pick-up-embedding-data-from-bert-model/14451)

[19] [Get embedding from finetuned BertForSequenceClassification model, HuggingFace discussion, 2022](https://discuss.huggingface.co/t/get-embedding-from-finetuned-bertforsequenceclassification-model/14897)

[20] [Sentence Embeddings From Fine-Tuned BERTForSequenceClassification, 2021](https://discuss.huggingface.co/t/sentence-embeddings-from-fine-tuned-bertforsequenceclassification/10345)

[21] [How to get [CLS] embeddings from BertForTokenClassification model, HuggingFace discussion, 2021](https://discuss.huggingface.co/t/how-to-get-cls-embeddings-from-bertfortokenclassification-model/9276)

[22] [transformers/Trainer on HuggingFace.co/docs, latest version](https://huggingface.co/docs/transformers/main_classes/trainer)

[23] [transformers/TrainingArguments on HuggingFace.co/docs, latest version](https://huggingface.co/docs/transformers/v4.51.3/en/main_classes/trainer#transformers.TrainingArguments)

[24] [Customized Evaluation Metrics with HuggingFace Trainer, R. Rajpurohit, Medium, 2023](https://medium.com/@rakeshrajpurohit/customized-evaluation-metrics-with-hugging-face-trainer-3ff00d936f99)

[25] [Index Error: Target is out-of-bounds, StackOverflow, 2022](https://stackoverflow.com/questions/70680290/indexerror-target-is-out-of-bounds)

[26] [NEFTune: Noisy Embeddings Improve Instruction Finetuning, N. Jain et al, 2023](https://arxiv.org/abs/2310.05914)

[27] <https://github.com/neelsjain/NEFTune>

[28] [Fine-tuning BERT model for arbitrarily long texts, Part 1, Michał Brzozowski, 2024](https://medium.com/mim-solutions-blog/fine-tuning-bert-model-for-arbitrarily-long-texts-part-1-299f1533b976)

[29] [Fine-tuning BERT model for arbitrarily long texts, Part 2, Michał Brzozowski, 2024](https://medium.com/mim-solutions-blog/fine-tuning-bert-model-for-arbitrarily-long-texts-part-2-3211d1774dc9)

# Appendix

## Detailed Results from the Training Phase

### Using the original query string as input text and the WineID as a class identifier

#### Using resampled aggregated dataset as evaluation set

click\_train = pd.read\_parquet(TRAIN\_DATASET\_PATH)

click\_test = pd.read\_parquet(TEST\_DATASET\_PATH)

catalog = pd.read\_parquet(CATALOG\_DATASET\_PATH)

click\_all = pd.concat([click\_train, click\_test], ignore\_index=True)

display(click\_all.head())

print(f"Number of rows in the training dataset: {len(click\_train)}")

print(f"Number of rows in the test dataset: {len(click\_test)}")

print(f"Number of rows in the combined dataset: {len(click\_all)}")

# split into 60% training, 20% test and 20% validation

click\_train\_new, click\_validate\_new, click\_test\_new = \

np.split(click\_all.sample(frac=1, random\_state=42),

[int(.6\*len(click\_all)), int(.8\*len(click\_all))])

Number of rows in the training dataset: 12932

Number of rows in the test dataset: 3233

Number of rows in the combined dataset: 16165

Number of rows in the resampled training dataset: 9699

**Number of rows in the resampled validation dataset: 3233**

Number of rows in the resampled test dataset: 3233

train\_texts = list(click\_train['query'])

val\_texts = list(click\_validate\_new['query'])

test\_texts = list(click\_test['query'])

train\_labels = list(click\_train['label'])

val\_labels = list(click\_validate\_new['label'])

test\_labels = list(click\_test['label'])

train\_encodings = tokenizer(train\_texts, truncation=True, padding=True)

val\_encodings = tokenizer(val\_texts, truncation=True, padding=True)

test\_encodings = tokenizer(test\_texts, truncation=True, padding=True)

train\_dataloader = DataLoader(train\_encodings, train\_labels)

val\_dataloader = DataLoader(val\_encodings, val\_labels)

test\_dataset = DataLoader(test\_encodings, test\_labels)

# Trainer

trainer = CustomTrainer(

model=model,

args=training\_args,

train\_dataset=train\_dataloader,

eval\_dataset=**val\_dataloader**,

compute\_metrics=compute\_metrics,

)

**[81000/81000 1:39:57, Epoch 200/200]**

| **Epoch** | **Training Loss** | **Validation Loss** | **Accuracy** | **F1** | **Precision** | **Recall** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 7.864300 | 7.821476 | 0.002784 | 0.000003 | 0.999439 | 0.000563 |
| 2 | 7.782100 | 7.696340 | 0.004949 | 0.000232 | 0.984623 | 0.004224 |
| 3 | 7.661000 | 7.543985 | 0.011444 | 0.001401 | 0.975259 | 0.007629 |
| 4 | 7.524600 | 7.387143 | 0.022580 | 0.005444 | 0.948202 | 0.019918 |
| 5 | 7.380200 | 7.231980 | 0.030003 | 0.007903 | 0.947600 | 0.022348 |
| 6 | 7.235000 | 7.083477 | 0.038045 | 0.009946 | 0.935086 | 0.030326 |
| 7 | 7.087900 | 6.930125 | 0.045159 | 0.013584 | 0.935400 | 0.035777 |
| 8 | 6.937200 | 6.782388 | 0.054129 | 0.017536 | 0.926495 | 0.042612 |
| 9 | 6.792200 | 6.639849 | 0.055057 | 0.019752 | 0.933639 | 0.042010 |
| 10 | 6.648600 | 6.502372 | 0.060625 | 0.023622 | 0.926870 | 0.046515 |
| 11 | 6.511800 | 6.365452 | 0.069595 | 0.027945 | 0.924597 | 0.053761 |
| 12 | 6.381000 | 6.242392 | 0.072688 | 0.029404 | 0.920449 | 0.057599 |
| 13 | 6.251000 | 6.115745 | 0.078874 | 0.034978 | 0.918269 | 0.063660 |
| 14 | 6.126400 | 5.998829 | 0.081658 | 0.034763 | 0.915068 | 0.065540 |
| 15 | 6.005000 | 5.885547 | 0.088463 | 0.039090 | 0.914194 | 0.073225 |
| 16 | 5.895900 | 5.778150 | 0.097123 | 0.043265 | 0.913201 | 0.077669 |
| 17 | 5.785400 | 5.676041 | 0.099907 | 0.045018 | 0.910979 | 0.080767 |
| 18 | 5.682800 | 5.580103 | 0.107949 | 0.050514 | 0.904999 | 0.088027 |
| 19 | 5.580000 | 5.481562 | 0.115373 | 0.055031 | 0.901790 | 0.094966 |
| 20 | 5.487000 | 5.394043 | 0.118775 | 0.060969 | 0.899526 | 0.098983 |
| 21 | 5.392500 | 5.308720 | 0.126817 | 0.068910 | 0.886766 | 0.109808 |
| 22 | 5.302700 | 5.225064 | 0.133003 | 0.071338 | 0.889937 | 0.113481 |
| 23 | 5.220300 | 5.141930 | 0.145994 | 0.078807 | 0.886938 | 0.124342 |
| 24 | 5.135600 | 5.066474 | 0.154346 | 0.085093 | 0.886365 | 0.129691 |
| 25 | 5.052900 | 4.992687 | 0.163316 | 0.091304 | 0.879228 | 0.140012 |
| 26 | 4.978600 | 4.916536 | 0.170430 | 0.096797 | 0.877800 | 0.147013 |
| 27 | 4.904800 | 4.851599 | 0.175998 | 0.100673 | 0.866946 | 0.152964 |
| 28 | 4.830600 | 4.779248 | 0.184040 | 0.108196 | 0.870683 | 0.159171 |
| 29 | 4.758700 | 4.714800 | 0.193938 | 0.116813 | 0.867532 | 0.168677 |
| 30 | 4.690900 | 4.653963 | 0.204145 | 0.124861 | 0.860574 | 0.179664 |
| 31 | 4.625200 | 4.585806 | 0.215589 | 0.132684 | 0.853300 | 0.191295 |
| 32 | 4.556500 | 4.527489 | 0.225487 | 0.141354 | 0.849745 | 0.202460 |
| 33 | 4.487400 | 4.469368 | 0.231364 | 0.144923 | 0.845200 | 0.206673 |
| 34 | 4.428400 | 4.406553 | 0.239715 | 0.155300 | 0.849145 | 0.218424 |
| 35 | 4.365900 | 4.346149 | 0.249613 | 0.164034 | 0.842092 | 0.227369 |
| 36 | 4.301800 | 4.294168 | 0.255800 | 0.165884 | 0.844393 | 0.229873 |
| 37 | 4.240100 | 4.236045 | 0.261367 | 0.173310 | 0.839105 | 0.239786 |
| 38 | 4.181100 | 4.183367 | 0.266625 | 0.175824 | 0.834362 | 0.241332 |
| 39 | 4.118800 | 4.125309 | 0.279307 | 0.183809 | 0.831217 | 0.255508 |
| 40 | 4.065400 | 4.076340 | 0.287968 | 0.192473 | 0.825307 | 0.265080 |
| 41 | 4.005700 | 4.022471 | 0.301268 | 0.204764 | 0.820440 | 0.278725 |
| 42 | 3.947400 | 3.973727 | 0.308073 | 0.211200 | 0.826342 | 0.284697 |
| 43 | 3.893700 | 3.917942 | 0.319827 | 0.221612 | 0.825492 | 0.295457 |
| 44 | 3.843100 | 3.872597 | 0.330034 | 0.227889 | 0.819674 | 0.303997 |
| 45 | 3.780700 | 3.822325 | 0.342406 | 0.240732 | 0.815999 | 0.320456 |
| 46 | 3.722600 | 3.775195 | 0.350758 | 0.248576 | 0.813635 | 0.327348 |
| 47 | 3.674500 | 3.724032 | 0.358491 | 0.255473 | 0.810616 | 0.337895 |
| 48 | 3.617900 | 3.680412 | 0.368079 | 0.262542 | 0.804102 | 0.350181 |
| 49 | 3.561700 | 3.633020 | 0.373956 | 0.268824 | 0.810003 | 0.353123 |
| 50 | 3.517400 | 3.582876 | 0.388494 | 0.277796 | 0.803548 | 0.365600 |
| 51 | 3.463500 | 3.538777 | 0.399010 | 0.289504 | 0.809180 | 0.373420 |
| 52 | 3.410100 | 3.488921 | 0.415094 | 0.300433 | 0.803139 | 0.388869 |
| 53 | 3.364000 | 3.446387 | 0.421899 | 0.308348 | 0.801097 | 0.399073 |
| 54 | 3.308400 | 3.407146 | 0.427776 | 0.315718 | 0.800994 | 0.405497 |
| 55 | 3.259500 | 3.356979 | 0.437674 | 0.322989 | 0.795642 | 0.416195 |
| 56 | 3.211600 | 3.317713 | 0.446025 | 0.328996 | 0.798452 | 0.420388 |
| 57 | 3.157600 | 3.273821 | 0.460254 | 0.342637 | 0.800632 | 0.435924 |
| 58 | 3.110300 | 3.231664 | 0.469224 | 0.356085 | 0.790581 | 0.446738 |
| 59 | 3.061800 | 3.188758 | 0.477884 | 0.362429 | 0.793284 | 0.456587 |
| 60 | 3.009500 | 3.148007 | 0.489638 | 0.376314 | 0.793845 | 0.468654 |
| 61 | 2.966700 | 3.109711 | 0.494278 | 0.379501 | 0.792505 | 0.472977 |
| 62 | 2.919300 | 3.068309 | 0.499227 | 0.382873 | 0.791958 | 0.478755 |
| 63 | 2.868700 | 3.024018 | 0.519023 | 0.403706 | 0.792430 | 0.498329 |
| 64 | 2.822500 | 2.991061 | 0.517476 | 0.403419 | 0.792867 | 0.498939 |
| 65 | 2.777400 | 2.947837 | 0.534179 | 0.421040 | 0.791833 | 0.517766 |
| 66 | 2.726200 | 2.915081 | 0.540674 | 0.425611 | 0.797662 | 0.519490 |
| 67 | 2.684500 | 2.876953 | 0.551809 | 0.437935 | 0.797164 | 0.531646 |
| 68 | 2.639100 | 2.840749 | 0.559542 | 0.446308 | 0.800873 | 0.540360 |
| 69 | 2.591300 | 2.801469 | 0.570059 | 0.461003 | 0.797194 | 0.550989 |
| 70 | 2.547900 | 2.767900 | 0.579647 | 0.469057 | 0.796260 | 0.561601 |
| 71 | 2.507500 | 2.730814 | 0.593876 | 0.481640 | 0.801167 | 0.577480 |
| 72 | 2.464100 | 2.698873 | 0.600680 | 0.486506 | 0.800474 | 0.582620 |
| 73 | 2.420500 | 2.666752 | 0.607176 | 0.490882 | 0.799906 | 0.588484 |
| 74 | 2.381100 | 2.631350 | 0.611197 | 0.495361 | 0.798186 | 0.594693 |
| 75 | 2.335500 | 2.595870 | 0.623879 | 0.512270 | 0.802995 | 0.609938 |
| 76 | 2.296200 | 2.566156 | 0.631612 | 0.519965 | 0.801256 | 0.616213 |
| 77 | 2.256400 | 2.537648 | 0.634086 | 0.523613 | 0.807635 | 0.619382 |
| 78 | 2.220500 | 2.500364 | 0.642747 | 0.528406 | 0.801933 | 0.628220 |
| 79 | 2.174900 | 2.471609 | 0.648933 | 0.537678 | 0.808641 | 0.632143 |
| 80 | 2.137500 | 2.442599 | 0.654191 | 0.541306 | 0.804218 | 0.639193 |
| 81 | 2.096400 | 2.412375 | 0.660996 | 0.549660 | 0.805669 | 0.644395 |
| 82 | 2.059500 | 2.383735 | 0.662852 | 0.553107 | 0.809337 | 0.647778 |
| 83 | 2.021900 | 2.356759 | 0.671203 | 0.561932 | 0.811495 | 0.656156 |
| 84 | 1.984000 | 2.328458 | 0.678627 | 0.572691 | 0.811401 | 0.666204 |
| 85 | 1.945800 | 2.299317 | 0.684504 | 0.577508 | 0.811944 | 0.669858 |
| 86 | 1.909400 | 2.274493 | 0.684504 | 0.578795 | 0.807921 | 0.669248 |
| 87 | 1.881100 | 2.245907 | 0.689453 | 0.585232 | 0.810789 | 0.674546 |
| 88 | 1.840700 | 2.223400 | 0.697185 | 0.593713 | 0.816867 | 0.682355 |
| 89 | 1.805400 | 2.195514 | 0.696567 | 0.591882 | 0.816978 | 0.680641 |
| 90 | 1.768300 | 2.177090 | 0.702134 | 0.600758 | 0.815718 | 0.688538 |
| 91 | 1.736000 | 2.153873 | 0.707702 | 0.607322 | 0.820019 | 0.695353 |
| 92 | 1.707100 | 2.129835 | 0.710486 | 0.610302 | 0.819148 | 0.698083 |
| 93 | 1.678500 | 2.105259 | 0.715435 | 0.614947 | 0.818998 | 0.704235 |
| 94 | 1.644800 | 2.083233 | 0.720384 | 0.619896 | 0.818494 | 0.710115 |
| 95 | 1.607500 | 2.062459 | 0.721621 | 0.624752 | 0.820278 | 0.711164 |
| 96 | 1.579000 | 2.041199 | 0.728116 | 0.633618 | 0.822989 | 0.719648 |
| 97 | 1.549000 | 2.022415 | 0.732447 | 0.637459 | 0.824175 | 0.725475 |
| 98 | 1.523500 | 2.002409 | 0.729972 | 0.632988 | 0.822966 | 0.721687 |
| 99 | 1.493500 | 1.982971 | 0.738014 | 0.642920 | 0.826202 | 0.731011 |
| 100 | 1.460500 | 1.962548 | 0.739561 | 0.645849 | 0.824026 | 0.734282 |
| 101 | 1.435100 | 1.945907 | 0.740798 | 0.649730 | 0.828676 | 0.735684 |
| 102 | 1.409400 | 1.928557 | 0.744200 | 0.654645 | 0.828228 | 0.739225 |
| 103 | 1.380100 | 1.907897 | 0.743273 | 0.651828 | 0.823002 | 0.738034 |
| 104 | 1.354700 | 1.891374 | 0.747294 | 0.656792 | 0.829094 | 0.741194 |
| 105 | 1.329000 | 1.872113 | 0.748221 | 0.656352 | 0.825444 | 0.744299 |
| 106 | 1.301300 | 1.858119 | 0.750387 | 0.658681 | 0.825660 | 0.747385 |
| 107 | 1.281300 | 1.845404 | 0.752861 | 0.662648 | 0.829543 | 0.747778 |
| 108 | 1.256500 | 1.832087 | 0.756882 | 0.668921 | 0.828057 | 0.754809 |
| 109 | 1.233900 | 1.813154 | 0.756573 | 0.667482 | 0.829698 | 0.753778 |
| 110 | 1.210000 | 1.801449 | 0.756264 | 0.666991 | 0.829191 | 0.752569 |
| 111 | 1.182800 | 1.783506 | 0.756573 | 0.670783 | 0.829969 | 0.754604 |
| 112 | 1.160200 | 1.773535 | 0.760903 | 0.673582 | 0.828882 | 0.759467 |
| 113 | 1.139600 | 1.764531 | 0.760903 | 0.673829 | 0.829076 | 0.760563 |
| 114 | 1.121500 | 1.745754 | 0.762140 | 0.674839 | 0.830081 | 0.759412 |
| 115 | 1.100200 | 1.734089 | 0.764924 | 0.679245 | 0.833394 | 0.764094 |
| 116 | 1.077600 | 1.724558 | 0.764615 | 0.677288 | 0.827699 | 0.764091 |
| 117 | 1.061900 | 1.711528 | 0.767399 | 0.680060 | 0.829923 | 0.766889 |
| 118 | 1.035500 | 1.700131 | 0.765543 | 0.678825 | 0.829246 | 0.765640 |
| 119 | 1.021000 | 1.689756 | 0.769255 | 0.686167 | 0.832726 | 0.768667 |
| 120 | 1.003200 | 1.678868 | 0.770182 | 0.684084 | 0.828437 | 0.771373 |
| 121 | 0.985200 | 1.668042 | 0.772657 | 0.688212 | 0.832664 | 0.771965 |
| 122 | 0.967600 | 1.656339 | 0.771729 | 0.686214 | 0.829547 | 0.772905 |
| 123 | 0.948400 | 1.650546 | 0.773585 | 0.688289 | 0.829909 | 0.774679 |
| 124 | 0.932000 | 1.639691 | 0.773585 | 0.690369 | 0.832538 | 0.775511 |
| 125 | 0.915000 | 1.629775 | 0.776059 | 0.691720 | 0.831272 | 0.776822 |
| 126 | 0.897900 | 1.623070 | 0.777606 | 0.695156 | 0.836498 | 0.777358 |
| 127 | 0.885100 | 1.612675 | 0.777297 | 0.693839 | 0.832431 | 0.778569 |
| 128 | 0.866900 | 1.604223 | 0.777297 | 0.694191 | 0.833644 | 0.777214 |
| 129 | 0.853200 | 1.598824 | 0.775131 | 0.689260 | 0.830552 | 0.775670 |
| 130 | 0.838800 | 1.588995 | 0.776369 | 0.691716 | 0.831361 | 0.776810 |
| 131 | 0.826500 | 1.581385 | 0.779152 | 0.692540 | 0.833508 | 0.778391 |
| 132 | 0.808500 | 1.577011 | 0.776987 | 0.690601 | 0.829305 | 0.777859 |
| 133 | 0.796000 | 1.568413 | 0.778225 | 0.692077 | 0.829784 | 0.780208 |
| 134 | 0.785700 | 1.564294 | 0.778843 | 0.693382 | 0.831023 | 0.779011 |
| 135 | 0.775800 | 1.556564 | 0.778843 | 0.694384 | 0.831397 | 0.781293 |
| 136 | 0.759900 | 1.548928 | 0.780080 | 0.694753 | 0.832224 | 0.782243 |
| 137 | 0.746100 | 1.543070 | 0.780699 | 0.695314 | 0.831486 | 0.782907 |
| 138 | 0.739100 | 1.541680 | 0.779462 | 0.695002 | 0.830221 | 0.781122 |
| 139 | 0.727900 | 1.533715 | 0.780080 | 0.695584 | 0.831170 | 0.782582 |
| 140 | 0.710900 | 1.529258 | 0.780699 | 0.696019 | 0.830393 | 0.783462 |
| 141 | 0.706500 | 1.523750 | 0.781318 | 0.697318 | 0.832045 | 0.784386 |
| 142 | 0.691000 | 1.518563 | 0.780390 | 0.695306 | 0.829919 | 0.784295 |
| 143 | 0.680000 | 1.514247 | 0.781318 | 0.696361 | 0.829883 | 0.784756 |
| 144 | 0.674800 | 1.510104 | 0.781627 | 0.697458 | 0.831698 | 0.785021 |
| 145 | 0.661500 | 1.504016 | 0.781627 | 0.696162 | 0.829708 | 0.784208 |
| 146 | 0.654500 | 1.501232 | 0.781318 | 0.696524 | 0.830791 | 0.784580 |
| 147 | 0.643500 | 1.495959 | 0.782246 | 0.697101 | 0.829375 | 0.786027 |
| 148 | 0.635400 | 1.494968 | 0.782246 | 0.697912 | 0.829208 | 0.786378 |
| 149 | 0.628000 | 1.490718 | 0.782246 | 0.699009 | 0.831009 | 0.786405 |
| 150 | 0.619700 | 1.486889 | 0.783174 | 0.698212 | 0.831143 | 0.786507 |
| 151 | 0.610400 | 1.483698 | 0.784720 | 0.699839 | 0.832406 | 0.788950 |
| 152 | 0.600700 | 1.480033 | 0.783792 | 0.699791 | 0.829512 | 0.787891 |
| 153 | 0.595800 | 1.474985 | 0.785339 | 0.702298 | 0.832347 | 0.790247 |
| 154 | 0.587000 | 1.473130 | 0.785029 | 0.700576 | 0.829784 | 0.788871 |
| 155 | 0.580000 | 1.470267 | 0.786576 | 0.701317 | 0.829132 | 0.790141 |
| 156 | 0.573200 | 1.466656 | 0.785029 | 0.700418 | 0.829514 | 0.789456 |
| 157 | 0.567300 | 1.465384 | 0.784720 | 0.702062 | 0.830461 | 0.789691 |
| 158 | 0.560700 | 1.461699 | 0.785339 | 0.702235 | 0.830079 | 0.790284 |
| 159 | 0.553300 | 1.458765 | 0.788122 | 0.706267 | 0.834575 | 0.792772 |
| 160 | 0.548800 | 1.458347 | 0.787195 | 0.702574 | 0.829053 | 0.791751 |
| 161 | 0.544900 | 1.455268 | 0.786267 | 0.705074 | 0.831874 | 0.790497 |
| 162 | 0.536000 | 1.454693 | 0.785957 | 0.702249 | 0.830842 | 0.791263 |
| 163 | 0.531700 | 1.452044 | 0.786576 | 0.704558 | 0.831814 | 0.791484 |
| 164 | 0.526200 | 1.449690 | 0.787813 | 0.704391 | 0.830831 | 0.792844 |
| 165 | 0.520200 | 1.446332 | 0.788432 | 0.706656 | 0.832794 | 0.793530 |
| 166 | 0.515200 | 1.446787 | 0.787813 | 0.706472 | 0.833323 | 0.793208 |
| 167 | 0.510000 | 1.443982 | 0.786885 | 0.706148 | 0.833523 | 0.792049 |
| 168 | 0.508300 | 1.441762 | 0.789978 | 0.707526 | 0.832506 | 0.794672 |
| 169 | 0.502500 | 1.441692 | 0.789360 | 0.707863 | 0.834713 | 0.793975 |
| 170 | 0.497400 | 1.439563 | 0.788432 | 0.706794 | 0.833053 | 0.793890 |
| 171 | 0.493600 | 1.439037 | 0.790906 | 0.709421 | 0.833605 | 0.796298 |
| 172 | 0.486400 | 1.437205 | 0.790288 | 0.708580 | 0.835051 | 0.795424 |
| 173 | 0.482900 | 1.436223 | 0.789050 | 0.706757 | 0.832895 | 0.793700 |
| 174 | 0.480900 | 1.435473 | 0.788432 | 0.707738 | 0.833236 | 0.793849 |
| 175 | 0.476700 | 1.434337 | 0.789360 | 0.707794 | 0.832234 | 0.795056 |
| 176 | 0.474100 | 1.433027 | 0.789050 | 0.707241 | 0.834259 | 0.794277 |
| 177 | 0.472000 | 1.431239 | 0.789978 | 0.709460 | 0.835208 | 0.795115 |
| 178 | 0.469700 | 1.429756 | 0.789978 | 0.708178 | 0.833692 | 0.795615 |
| 179 | 0.462100 | 1.430295 | 0.789360 | 0.708270 | 0.834589 | 0.793977 |
| 180 | 0.463700 | 1.429035 | 0.789669 | 0.707315 | 0.832990 | 0.794842 |
| 181 | 0.454800 | 1.427691 | 0.790288 | 0.709657 | 0.834358 | 0.794913 |
| 182 | 0.456200 | 1.426998 | 0.789669 | 0.708879 | 0.833956 | 0.794889 |
| 183 | 0.453600 | 1.425848 | 0.789669 | 0.709451 | 0.834146 | 0.794629 |
| 184 | 0.450300 | 1.425220 | 0.790288 | 0.710182 | 0.835038 | 0.795819 |
| 185 | 0.451300 | 1.424901 | 0.789050 | 0.708332 | 0.833256 | 0.794324 |
| 186 | 0.446600 | 1.423585 | 0.789360 | 0.708258 | 0.831074 | 0.794696 |
| 187 | 0.445100 | 1.422845 | 0.789978 | 0.709486 | 0.835148 | 0.794696 |
| 188 | 0.446100 | 1.422781 | 0.789978 | 0.710174 | 0.834379 | 0.795004 |
| 189 | 0.440800 | 1.423176 | 0.789669 | 0.709554 | 0.833943 | 0.794476 |
| 190 | 0.438800 | 1.422529 | 0.789669 | 0.709013 | 0.834619 | 0.794825 |
| 191 | 0.439100 | 1.422366 | 0.789360 | 0.708974 | 0.833859 | 0.794321 |
| 192 | 0.437700 | 1.421920 | 0.789360 | 0.710491 | 0.834761 | 0.794905 |
| 193 | 0.434900 | 1.421387 | 0.789360 | 0.709098 | 0.833349 | 0.795010 |
| 194 | 0.434900 | 1.420646 | 0.790288 | 0.709263 | 0.833700 | 0.795700 |
| 195 | 0.432600 | 1.420850 | 0.789978 | 0.710081 | 0.833893 | 0.795271 |
| 196 | 0.431900 | 1.420529 | 0.789360 | 0.709704 | 0.833623 | 0.795104 |
| 197 | 0.432000 | 1.420300 | 0.789669 | 0.709647 | 0.833676 | 0.795186 |
| 198 | 0.429600 | 1.420308 | 0.789978 | 0.709749 | 0.833544 | 0.795361 |
| 199 | 0.430500 | 1.420155 | 0.789360 | 0.709524 | 0.833588 | 0.795010 |
| 200 | 0.428300 | 1.420119 | 0.789669 | 0.709872 | 0.834004 | 0.795186 |

TrainOutput(global\_step=81000, training\_loss=2.2710581553894795, metrics={'train\_runtime': 5997.4878, 'train\_samples\_per\_second': 431.247, 'train\_steps\_per\_second': 13.506, 'total\_flos': 4.62442691165952e+16, 'train\_loss': 2.2710581553894795, 'epoch': 200.0})

### Using compound query string as input text and the WineID as a class identifier

Recall, in this case instead of the original query string the new compound query will be a concatenation of the original query string and the input converted to string. In order to limit the number of tokens generated by the tokenizer we limit the number of appended wine ids from the results field to 8.

**[60750/60750 1:21:19, Epoch 150/150]**

| **Epoch** | **Training Loss** | **Validation Loss** | **Accuracy** | **F1** | **Precision** | **Recall** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 7.861100 | 7.854743 | 0.000619 | 0.000001 | 0.999448 | 0.000552 |
| 2 | 7.780900 | 7.814166 | 0.001856 | 0.000019 | 0.990670 | 0.005262 |
| 3 | 7.667600 | 7.698745 | 0.006805 | 0.000436 | 0.976247 | 0.009653 |
| 4 | 7.528400 | 7.578595 | 0.014228 | 0.001044 | 0.964003 | 0.017615 |
| 5 | 7.383000 | 7.457574 | 0.019177 | 0.003405 | 0.952381 | 0.020406 |
| 6 | 7.239900 | 7.346857 | 0.020724 | 0.003853 | 0.951775 | 0.021584 |
| 7 | 7.094500 | 7.233721 | 0.023817 | 0.005175 | 0.948312 | 0.025615 |
| 8 | 6.955600 | 7.138432 | 0.025982 | 0.005068 | 0.942157 | 0.028393 |
| 9 | 6.825800 | 7.041925 | 0.025673 | 0.005359 | 0.944146 | 0.027296 |
| 10 | 6.688500 | 6.938967 | 0.029694 | 0.006650 | 0.938955 | 0.030502 |
| 11 | 6.560300 | 6.846939 | 0.030931 | 0.007870 | 0.932175 | 0.033023 |
| 12 | 6.438900 | 6.751650 | 0.035261 | 0.010770 | 0.926526 | 0.037657 |
| 13 | 6.314800 | 6.673194 | 0.037117 | 0.011869 | 0.926625 | 0.042036 |
| 14 | 6.201800 | 6.596253 | 0.040520 | 0.011765 | 0.916430 | 0.045039 |
| 15 | 6.082700 | 6.523460 | 0.042066 | 0.012484 | 0.916470 | 0.046842 |
| 16 | 5.973700 | 6.452928 | 0.044850 | 0.013982 | 0.910861 | 0.051956 |
| 17 | 5.868500 | 6.391502 | 0.047634 | 0.015002 | 0.908510 | 0.055107 |
| 18 | 5.764300 | 6.336549 | 0.050727 | 0.017963 | 0.909632 | 0.056216 |
| 19 | 5.666800 | 6.270074 | 0.050727 | 0.017800 | 0.898094 | 0.060617 |
| 20 | 5.570800 | 6.204684 | 0.052273 | 0.017418 | 0.888358 | 0.063112 |
| 21 | 5.475500 | 6.154991 | 0.057222 | 0.020716 | 0.884803 | 0.071535 |
| 22 | 5.379500 | 6.109480 | 0.055676 | 0.021295 | 0.879236 | 0.070043 |
| 23 | 5.291700 | 6.061592 | 0.056294 | 0.019297 | 0.871078 | 0.072177 |
| 24 | 5.197700 | 6.015542 | 0.056913 | 0.021955 | 0.868455 | 0.075205 |
| 25 | 5.115900 | 5.968026 | 0.059078 | 0.021763 | 0.857426 | 0.081234 |
| 26 | 5.035300 | 5.926356 | 0.061553 | 0.023782 | 0.853774 | 0.084458 |
| 27 | 4.951000 | 5.889604 | 0.058769 | 0.022869 | 0.857537 | 0.081167 |
| 28 | 4.873600 | 5.860871 | 0.062171 | 0.024523 | 0.846567 | 0.088976 |
| 29 | 4.795900 | 5.820742 | 0.064646 | 0.026573 | 0.838707 | 0.092404 |
| 30 | 4.718000 | 5.784391 | 0.065264 | 0.027487 | 0.830799 | 0.095433 |
| 31 | 4.640100 | 5.745276 | 0.066502 | 0.027665 | 0.819656 | 0.101950 |
| 32 | 4.568500 | 5.711545 | 0.068048 | 0.029250 | 0.819072 | 0.102365 |
| 33 | 4.491000 | 5.695155 | 0.069904 | 0.029557 | 0.819173 | 0.106756 |
| 34 | 4.422500 | 5.665586 | 0.070213 | 0.030596 | 0.808549 | 0.110451 |
| 35 | 4.354100 | 5.636877 | 0.071760 | 0.032473 | 0.803713 | 0.112347 |
| 36 | 4.280400 | 5.620381 | 0.070832 | 0.031083 | 0.799327 | 0.111246 |
| 37 | 4.215200 | 5.582979 | 0.074234 | 0.034186 | 0.786658 | 0.118660 |
| 38 | 4.146900 | 5.561170 | 0.077018 | 0.037980 | 0.782103 | 0.123951 |
| 39 | 4.078700 | 5.541314 | 0.078255 | 0.040192 | 0.775637 | 0.127183 |
| 40 | 4.013600 | 5.528586 | 0.080730 | 0.041355 | 0.767751 | 0.133540 |
| 41 | 3.950000 | 5.496936 | 0.083204 | 0.041536 | 0.762795 | 0.134477 |
| 42 | 3.887900 | 5.491465 | 0.081658 | 0.042860 | 0.759567 | 0.134630 |
| 43 | 3.819800 | 5.473618 | 0.084442 | 0.044576 | 0.758152 | 0.139211 |
| 44 | 3.759600 | 5.451656 | 0.084442 | 0.044103 | 0.737780 | 0.144467 |
| 45 | 3.693700 | 5.432120 | 0.088463 | 0.047058 | 0.736958 | 0.148524 |
| 46 | 3.634700 | 5.417769 | 0.088463 | 0.046640 | 0.733025 | 0.148332 |
| 47 | 3.576500 | 5.404928 | 0.088463 | 0.045130 | 0.723086 | 0.149734 |
| 48 | 3.511700 | 5.385719 | 0.088772 | 0.047405 | 0.719799 | 0.151768 |
| 49 | 3.453300 | 5.370276 | 0.095577 | 0.052110 | 0.721319 | 0.161520 |
| 50 | 3.401600 | 5.369367 | 0.091865 | 0.048673 | 0.712276 | 0.156494 |
| 51 | 3.341500 | 5.356624 | 0.094649 | 0.050577 | 0.706926 | 0.162540 |
| 52 | 3.279900 | 5.346323 | 0.098361 | 0.054759 | 0.702176 | 0.167133 |
| 53 | 3.231600 | 5.330283 | 0.096505 | 0.050927 | 0.695368 | 0.166630 |
| 54 | 3.167800 | 5.320364 | 0.097433 | 0.054305 | 0.686228 | 0.170411 |
| 55 | 3.115900 | 5.303611 | 0.098670 | 0.054945 | 0.687256 | 0.174984 |
| 56 | 3.058100 | 5.301453 | 0.100835 | 0.056345 | 0.672129 | 0.179217 |
| 57 | 3.006300 | 5.293799 | 0.102072 | 0.056976 | 0.670208 | 0.180487 |
| 58 | 2.951000 | 5.286605 | 0.099907 | 0.055909 | 0.668973 | 0.176765 |
| 59 | 2.900400 | 5.283617 | 0.101763 | 0.058288 | 0.655375 | 0.180822 |
| 60 | 2.849600 | 5.275704 | 0.101763 | 0.057627 | 0.655932 | 0.183260 |
| 61 | 2.796300 | 5.259170 | 0.104238 | 0.059796 | 0.650085 | 0.187749 |
| 62 | 2.751100 | 5.258082 | 0.107021 | 0.062771 | 0.648467 | 0.191597 |
| 63 | 2.698800 | 5.265430 | 0.104547 | 0.061861 | 0.641037 | 0.191736 |
| 64 | 2.649600 | 5.252347 | 0.104547 | 0.062247 | 0.639698 | 0.192074 |
| 65 | 2.602200 | 5.244664 | 0.105784 | 0.063374 | 0.629834 | 0.192687 |
| 66 | 2.553000 | 5.231055 | 0.103619 | 0.062664 | 0.618081 | 0.195354 |
| 67 | 2.505700 | 5.233306 | 0.105165 | 0.062826 | 0.612466 | 0.198261 |
| 68 | 2.459400 | 5.221867 | 0.108259 | 0.064911 | 0.605033 | 0.205868 |
| 69 | 2.416000 | 5.227592 | 0.109187 | 0.065859 | 0.611180 | 0.202672 |
| 70 | 2.367500 | 5.218062 | 0.111970 | 0.066500 | 0.608149 | 0.206524 |
| 71 | 2.327200 | 5.216759 | 0.110733 | 0.068592 | 0.604968 | 0.201167 |
| 72 | 2.282000 | 5.211370 | 0.109805 | 0.066369 | 0.600736 | 0.207094 |
| 73 | 2.243600 | 5.212341 | 0.111661 | 0.069229 | 0.605122 | 0.206796 |
| 74 | 2.197300 | 5.207943 | 0.112898 | 0.069380 | 0.595617 | 0.209641 |
| 75 | 2.158500 | 5.203680 | 0.114135 | 0.070502 | 0.592760 | 0.216238 |
| 76 | 2.115500 | 5.200793 | 0.115063 | 0.071554 | 0.588955 | 0.217104 |
| 77 | 2.075200 | 5.193475 | 0.116610 | 0.071534 | 0.587887 | 0.217747 |
| 78 | 2.040600 | 5.199539 | 0.114445 | 0.071691 | 0.587991 | 0.219652 |
| 79 | 1.997100 | 5.200224 | 0.115063 | 0.072468 | 0.579731 | 0.220481 |
| 80 | 1.961300 | 5.196666 | 0.114754 | 0.072097 | 0.580808 | 0.221847 |
| 81 | 1.924800 | 5.195502 | 0.114135 | 0.071384 | 0.576817 | 0.222250 |
| 82 | 1.887300 | 5.192952 | 0.115063 | 0.073127 | 0.566740 | 0.225724 |
| 83 | 1.855500 | 5.189295 | 0.116610 | 0.073959 | 0.566452 | 0.229973 |
| 84 | 1.820700 | 5.186652 | 0.118466 | 0.074645 | 0.565153 | 0.229891 |
| 85 | 1.782400 | 5.188110 | 0.120322 | 0.075184 | 0.572678 | 0.229183 |
| 86 | 1.754800 | 5.183352 | 0.118775 | 0.076006 | 0.557505 | 0.231018 |
| 87 | 1.723400 | 5.186430 | 0.119394 | 0.076527 | 0.564355 | 0.232357 |
| 88 | 1.683000 | 5.183125 | 0.120940 | 0.077610 | 0.557040 | 0.235000 |
| 89 | 1.651700 | 5.179712 | 0.121250 | 0.077211 | 0.559573 | 0.237439 |
| 90 | 1.621700 | 5.191047 | 0.121868 | 0.078276 | 0.559222 | 0.235928 |
| 91 | 1.594700 | 5.183854 | 0.121250 | 0.077340 | 0.540772 | 0.236857 |
| 92 | 1.563500 | 5.187104 | 0.120322 | 0.077015 | 0.544935 | 0.237645 |
| 93 | 1.537400 | 5.193796 | 0.120631 | 0.078977 | 0.549666 | 0.234217 |
| 94 | 1.510500 | 5.187129 | 0.120322 | 0.076440 | 0.541229 | 0.236478 |
| 95 | 1.481200 | 5.186184 | 0.120322 | 0.077060 | 0.536008 | 0.239086 |
| 96 | 1.454900 | 5.176236 | 0.120012 | 0.076294 | 0.535472 | 0.242215 |
| 97 | 1.430100 | 5.192218 | 0.123105 | 0.080142 | 0.538749 | 0.240960 |
| 98 | 1.408500 | 5.189687 | 0.120012 | 0.075212 | 0.536021 | 0.238178 |
| 99 | 1.382800 | 5.180948 | 0.122796 | 0.077660 | 0.523697 | 0.245155 |
| 100 | 1.354500 | 5.181475 | 0.123415 | 0.078085 | 0.535591 | 0.242073 |
| 101 | 1.334200 | 5.183085 | 0.124343 | 0.079064 | 0.530113 | 0.246150 |
| 102 | 1.310000 | 5.180462 | 0.125889 | 0.079353 | 0.524687 | 0.246860 |
| 103 | 1.289400 | 5.191702 | 0.124033 | 0.078932 | 0.524108 | 0.246097 |
| 104 | 1.268800 | 5.181412 | 0.123105 | 0.079839 | 0.514222 | 0.248034 |
| 105 | 1.245700 | 5.185920 | 0.123415 | 0.079389 | 0.518161 | 0.250395 |
| 106 | 1.225800 | 5.186183 | 0.124961 | 0.080709 | 0.517008 | 0.251497 |
| 107 | 1.208900 | 5.188387 | 0.123415 | 0.079166 | 0.508958 | 0.250320 |
| 108 | 1.187300 | 5.182910 | 0.127127 | 0.082443 | 0.515359 | 0.250655 |
| 109 | 1.168600 | 5.185389 | 0.124652 | 0.080119 | 0.507779 | 0.252919 |
| 110 | 1.149700 | 5.196910 | 0.124961 | 0.080314 | 0.512314 | 0.245985 |
| 111 | 1.133400 | 5.187031 | 0.124652 | 0.079960 | 0.510085 | 0.248992 |
| 112 | 1.114100 | 5.186946 | 0.123105 | 0.081048 | 0.511146 | 0.251415 |
| 113 | 1.099900 | 5.193588 | 0.126508 | 0.083186 | 0.511647 | 0.249833 |
| 114 | 1.083600 | 5.201047 | 0.125271 | 0.080203 | 0.510323 | 0.250146 |
| 115 | 1.067100 | 5.200179 | 0.124033 | 0.080461 | 0.510816 | 0.250169 |
| 116 | 1.052100 | 5.198524 | 0.124652 | 0.080171 | 0.505864 | 0.253934 |
| 117 | 1.039900 | 5.197755 | 0.124343 | 0.080300 | 0.505590 | 0.252646 |
| 118 | 1.024800 | 5.199993 | 0.127745 | 0.081447 | 0.506880 | 0.258718 |
| 119 | 1.008300 | 5.196416 | 0.127127 | 0.082311 | 0.497056 | 0.257881 |
| 120 | 0.999900 | 5.195811 | 0.123105 | 0.080260 | 0.496560 | 0.254383 |
| 121 | 0.984600 | 5.197592 | 0.127127 | 0.082373 | 0.493784 | 0.257334 |
| 122 | 0.973700 | 5.200027 | 0.127436 | 0.082618 | 0.496270 | 0.255763 |
| 123 | 0.961700 | 5.199306 | 0.126817 | 0.082501 | 0.499698 | 0.253706 |
| 124 | 0.951700 | 5.204721 | 0.127127 | 0.082843 | 0.495289 | 0.257877 |
| 125 | 0.937700 | 5.205031 | 0.126817 | 0.082179 | 0.499101 | 0.256598 |
| 126 | 0.928600 | 5.205366 | 0.125271 | 0.081618 | 0.496378 | 0.254170 |
| 127 | 0.919500 | 5.204683 | 0.128673 | 0.083231 | 0.497670 | 0.256469 |
| 128 | 0.912400 | 5.204875 | 0.126817 | 0.082664 | 0.489879 | 0.257047 |
| 129 | 0.902500 | 5.206861 | 0.128054 | 0.083593 | 0.493813 | 0.257986 |
| 130 | 0.891600 | 5.201454 | 0.128982 | 0.085394 | 0.487686 | 0.260948 |
| 131 | 0.884000 | 5.210475 | 0.126508 | 0.083100 | 0.489486 | 0.258117 |
| 132 | 0.875500 | 5.209958 | 0.128054 | 0.082726 | 0.484914 | 0.257587 |
| 133 | 0.873500 | 5.205686 | 0.129910 | 0.085457 | 0.488298 | 0.263885 |
| 134 | 0.865200 | 5.208210 | 0.129292 | 0.083949 | 0.487257 | 0.260423 |
| 135 | 0.858400 | 5.210942 | 0.129601 | 0.085490 | 0.489498 | 0.258995 |
| 136 | 0.852300 | 5.210137 | 0.128982 | 0.084338 | 0.485584 | 0.262668 |
| 137 | 0.845100 | 5.215831 | 0.126508 | 0.082319 | 0.481921 | 0.262177 |
| 138 | 0.840500 | 5.216609 | 0.128673 | 0.083676 | 0.485633 | 0.261023 |
| 139 | 0.833100 | 5.213460 | 0.128673 | 0.083592 | 0.481727 | 0.262943 |
| 140 | 0.828100 | 5.214323 | 0.128364 | 0.082992 | 0.479582 | 0.263118 |
| 141 | 0.827200 | 5.213132 | 0.127127 | 0.083717 | 0.480198 | 0.263311 |
| 142 | 0.820800 | 5.216689 | 0.128364 | 0.083734 | 0.477026 | 0.262474 |
| 143 | 0.818700 | 5.213425 | 0.128982 | 0.085098 | 0.480331 | 0.263050 |
| 144 | 0.819300 | 5.215407 | 0.130529 | 0.086071 | 0.481091 | 0.265066 |
| 145 | 0.812500 | 5.215313 | 0.129601 | 0.085207 | 0.480218 | 0.264403 |
| 146 | 0.812900 | 5.215034 | 0.128673 | 0.084903 | 0.478215 | 0.264399 |
| 147 | 0.809200 | 5.216928 | 0.129292 | 0.084652 | 0.481441 | 0.263702 |
| 148 | 0.805400 | 5.215889 | 0.130220 | 0.085416 | 0.483000 | 0.265099 |
| 149 | 0.808200 | 5.216158 | 0.129910 | 0.085048 | 0.481006 | 0.264264 |
| 150 | 0.803700 | 5.215966 | 0.130529 | 0.085578 | 0.481202 | 0.264804 |

TrainOutput(global\_step=60750, training\_loss=2.825293725284529, metrics={'train\_runtime': 4879.8661, 'train\_samples\_per\_second': 397.511, 'train\_steps\_per\_second': 12.449, 'total\_flos': 5.20248027561696e+16, 'train\_loss': 2.825293725284529, 'epoch': 150.0})

## Inside HF Transformers Roberta Model

<https://github.com/huggingface/transformers/blob/main/src/transformers/models/roberta/modeling_roberta.py>

### Roberta Embedding

class RobertaEmbeddings(nn.Module):

"""

Same as BertEmbeddings with a tiny tweak for positional embeddings indexing.

"""

# Copied from transformers.models.bert.modeling\_bert.BertEmbeddings.\_\_init\_\_

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.word\_embeddings = nn.Embedding(config.vocab\_size, config.hidden\_size, padding\_idx=config.pad\_token\_id)

self.position\_embeddings = nn.Embedding(config.max\_position\_embeddings, config.hidden\_size)

self.token\_type\_embeddings = nn.Embedding(config.type\_vocab\_size, config.hidden\_size)

# self.LayerNorm is not snake-cased to stick with TensorFlow model variable name and be able to load

# any TensorFlow checkpoint file

self.LayerNorm = nn.LayerNorm(config.hidden\_size, eps=config.layer\_norm\_eps)

self.dropout = nn.Dropout(config.hidden\_dropout\_prob)

# position\_ids (1, len position emb) is contiguous in memory and exported when serialized

self.position\_embedding\_type = getattr(config, "position\_embedding\_type", "absolute")

self.register\_buffer(

"position\_ids", torch.arange(config.max\_position\_embeddings).expand((1, -1)), persistent=False

)

self.register\_buffer(

"token\_type\_ids", torch.zeros(self.position\_ids.size(), dtype=torch.long), persistent=False

)

# End copy

self.padding\_idx = config.pad\_token\_id

self.position\_embeddings = nn.Embedding(

config.max\_position\_embeddings, config.hidden\_size, padding\_idx=self.padding\_idx

)

def forward(

self, input\_ids=None, token\_type\_ids=None, position\_ids=None, inputs\_embeds=None, past\_key\_values\_length=0

):

if position\_ids is None:

if input\_ids is not None:

# Create the position ids from the input token ids. Any padded tokens remain padded.

position\_ids = create\_position\_ids\_from\_input\_ids(input\_ids, self.padding\_idx, past\_key\_values\_length)

else:

position\_ids = self.create\_position\_ids\_from\_inputs\_embeds(inputs\_embeds)

if input\_ids is not None:

input\_shape = input\_ids.size()

else:

input\_shape = inputs\_embeds.size()[:-1]

seq\_length = input\_shape[1]

# Setting the token\_type\_ids to the registered buffer in constructor where it is all zeros, which usually occurs

# when its auto-generated, registered buffer helps users when tracing the model without passing token\_type\_ids, solves

# issue #5664

if token\_type\_ids is None:

if hasattr(self, "token\_type\_ids"):

buffered\_token\_type\_ids = self.token\_type\_ids[:, :seq\_length]

buffered\_token\_type\_ids\_expanded = buffered\_token\_type\_ids.expand(input\_shape[0], seq\_length)

token\_type\_ids = buffered\_token\_type\_ids\_expanded

else:

token\_type\_ids = torch.zeros(input\_shape, dtype=torch.long, device=self.position\_ids.device)

if inputs\_embeds is None:

inputs\_embeds = self.word\_embeddings(input\_ids)

token\_type\_embeddings = self.token\_type\_embeddings(token\_type\_ids)

embeddings = inputs\_embeds + token\_type\_embeddings

if self.position\_embedding\_type == "absolute":

position\_embeddings = self.position\_embeddings(position\_ids)

embeddings += position\_embeddings

embeddings = self.LayerNorm(embeddings)

embeddings = self.dropout(embeddings)

return embeddings

def create\_position\_ids\_from\_inputs\_embeds(self, inputs\_embeds):

"""

We are provided embeddings directly. We cannot infer which are padded so just generate sequential position ids.

Args:

inputs\_embeds: torch.Tensor

Returns: torch.Tensor

"""

input\_shape = inputs\_embeds.size()[:-1]

sequence\_length = input\_shape[1]

position\_ids = torch.arange(

self.padding\_idx + 1, sequence\_length + self.padding\_idx + 1, dtype=torch.long, device=inputs\_embeds.device

)

return position\_ids.unsqueeze(0).expand(input\_shape)

### Roberta Self Attention

# Copied from transformers.models.bert.modeling\_bert.BertSelfAttention with Bert->Roberta

class RobertaSelfAttention(nn.Module):

def \_\_init\_\_(self, config, position\_embedding\_type=None):

super().\_\_init\_\_()

if config.hidden\_size % config.num\_attention\_heads != 0 and not hasattr(config, "embedding\_size"):

raise ValueError(

f"The hidden size ({config.hidden\_size}) is not a multiple of the number of attention "

f"heads ({config.num\_attention\_heads})"

)

self.num\_attention\_heads = config.num\_attention\_heads

self.attention\_head\_size = int(config.hidden\_size / config.num\_attention\_heads)

self.all\_head\_size = self.num\_attention\_heads \* self.attention\_head\_size

self.query = nn.Linear(config.hidden\_size, self.all\_head\_size)

self.key = nn.Linear(config.hidden\_size, self.all\_head\_size)

self.value = nn.Linear(config.hidden\_size, self.all\_head\_size)

self.dropout = nn.Dropout(config.attention\_probs\_dropout\_prob)

self.position\_embedding\_type = position\_embedding\_type or getattr(

config, "position\_embedding\_type", "absolute"

)

if self.position\_embedding\_type == "relative\_key" or self.position\_embedding\_type == "relative\_key\_query":

self.max\_position\_embeddings = config.max\_position\_embeddings

self.distance\_embedding = nn.Embedding(2 \* config.max\_position\_embeddings - 1, self.attention\_head\_size)

self.is\_decoder = config.is\_decoder

def transpose\_for\_scores(self, x: torch.Tensor) -> torch.Tensor:

new\_x\_shape = x.size()[:-1] + (self.num\_attention\_heads, self.attention\_head\_size)

x = x.view(new\_x\_shape)

return x.permute(0, 2, 1, 3)

def forward(

self,

hidden\_states: torch.Tensor,

attention\_mask: Optional[torch.FloatTensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

encoder\_hidden\_states: Optional[torch.FloatTensor] = None,

encoder\_attention\_mask: Optional[torch.FloatTensor] = None,

past\_key\_value: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,

output\_attentions: Optional[bool] = False,

) -> Tuple[torch.Tensor]:

mixed\_query\_layer = self.query(hidden\_states)

# If this is instantiated as a cross-attention module, the keys

# and values come from an encoder; the attention mask needs to be

# such that the encoder's padding tokens are not attended to.

is\_cross\_attention = encoder\_hidden\_states is not None

if is\_cross\_attention and past\_key\_value is not None:

# reuse k,v, cross\_attentions

key\_layer = past\_key\_value[0]

value\_layer = past\_key\_value[1]

attention\_mask = encoder\_attention\_mask

elif is\_cross\_attention:

key\_layer = self.transpose\_for\_scores(self.key(encoder\_hidden\_states))

value\_layer = self.transpose\_for\_scores(self.value(encoder\_hidden\_states))

attention\_mask = encoder\_attention\_mask

elif past\_key\_value is not None:

key\_layer = self.transpose\_for\_scores(self.key(hidden\_states))

value\_layer = self.transpose\_for\_scores(self.value(hidden\_states))

key\_layer = torch.cat([past\_key\_value[0], key\_layer], dim=2)

value\_layer = torch.cat([past\_key\_value[1], value\_layer], dim=2)

else:

key\_layer = self.transpose\_for\_scores(self.key(hidden\_states))

value\_layer = self.transpose\_for\_scores(self.value(hidden\_states))

query\_layer = self.transpose\_for\_scores(mixed\_query\_layer)

use\_cache = past\_key\_value is not None

if self.is\_decoder:

# if cross\_attention save Tuple(torch.Tensor, torch.Tensor) of all cross attention key/value\_states.

# Further calls to cross\_attention layer can then reuse all cross-attention

# key/value\_states (first "if" case)

# if uni-directional self-attention (decoder) save Tuple(torch.Tensor, torch.Tensor) of

# all previous decoder key/value\_states. Further calls to uni-directional self-attention

# can concat previous decoder key/value\_states to current projected key/value\_states (third "elif" case)

# if encoder bi-directional self-attention `past\_key\_value` is always `None`

past\_key\_value = (key\_layer, value\_layer)

# Take the dot product between "query" and "key" to get the raw attention scores.

attention\_scores = torch.matmul(query\_layer, key\_layer.transpose(-1, -2))

if self.position\_embedding\_type == "relative\_key" or self.position\_embedding\_type == "relative\_key\_query":

query\_length, key\_length = query\_layer.shape[2], key\_layer.shape[2]

if use\_cache:

position\_ids\_l = torch.tensor(key\_length - 1, dtype=torch.long, device=hidden\_states.device).view(

-1, 1

)

else:

position\_ids\_l = torch.arange(query\_length, dtype=torch.long, device=hidden\_states.device).view(-1, 1)

position\_ids\_r = torch.arange(key\_length, dtype=torch.long, device=hidden\_states.device).view(1, -1)

distance = position\_ids\_l - position\_ids\_r

positional\_embedding = self.distance\_embedding(distance + self.max\_position\_embeddings - 1)

positional\_embedding = positional\_embedding.to(dtype=query\_layer.dtype) # fp16 compatibility

if self.position\_embedding\_type == "relative\_key":

relative\_position\_scores = torch.einsum("bhld,lrd->bhlr", query\_layer, positional\_embedding)

attention\_scores = attention\_scores + relative\_position\_scores

elif self.position\_embedding\_type == "relative\_key\_query":

relative\_position\_scores\_query = torch.einsum("bhld,lrd->bhlr", query\_layer, positional\_embedding)

relative\_position\_scores\_key = torch.einsum("bhrd,lrd->bhlr", key\_layer, positional\_embedding)

attention\_scores = attention\_scores + relative\_position\_scores\_query + relative\_position\_scores\_key

attention\_scores = attention\_scores / math.sqrt(self.attention\_head\_size)

if attention\_mask is not None:

# Apply the attention mask is (precomputed for all layers in RobertaModel forward() function)

attention\_scores = attention\_scores + attention\_mask

# Normalize the attention scores to probabilities.

attention\_probs = nn.functional.softmax(attention\_scores, dim=-1)

# This is actually dropping out entire tokens to attend to, which might

# seem a bit unusual, but is taken from the original Transformer paper.

attention\_probs = self.dropout(attention\_probs)

# Mask heads if we want to

if head\_mask is not None:

attention\_probs = attention\_probs \* head\_mask

context\_layer = torch.matmul(attention\_probs, value\_layer)

context\_layer = context\_layer.permute(0, 2, 1, 3).contiguous()

new\_context\_layer\_shape = context\_layer.size()[:-2] + (self.all\_head\_size,)

context\_layer = context\_layer.view(new\_context\_layer\_shape)

outputs = (context\_layer, attention\_probs) if output\_attentions else (context\_layer,)

if self.is\_decoder:

outputs = outputs + (past\_key\_value,)

return outputs

### Roberta Sdpa Self Attention

# Copied from transformers.models.bert.modeling\_bert.BertSdpaSelfAttention with Bert->Roberta

class RobertaSdpaSelfAttention(RobertaSelfAttention):

def \_\_init\_\_(self, config, position\_embedding\_type=None):

super().\_\_init\_\_(config, position\_embedding\_type=position\_embedding\_type)

self.dropout\_prob = config.attention\_probs\_dropout\_prob

self.require\_contiguous\_qkv = version.parse(get\_torch\_version()) < version.parse("2.2.0")

# Adapted from RobertaSelfAttention

def forward(

self,

hidden\_states: torch.Tensor,

attention\_mask: Optional[torch.Tensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

encoder\_hidden\_states: Optional[torch.FloatTensor] = None,

encoder\_attention\_mask: Optional[torch.FloatTensor] = None,

past\_key\_value: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,

output\_attentions: Optional[bool] = False,

) -> Tuple[torch.Tensor]:

if self.position\_embedding\_type != "absolute" or output\_attentions or head\_mask is not None:

# TODO: Improve this warning with e.g. `model.config.\_attn\_implementation = "manual"` once implemented.

logger.warning\_once(

"RobertaSdpaSelfAttention is used but `torch.nn.functional.scaled\_dot\_product\_attention` does not support "

"non-absolute `position\_embedding\_type` or `output\_attentions=True` or `head\_mask`. Falling back to "

"the manual attention implementation, but specifying the manual implementation will be required from "

"Transformers version v5.0.0 onwards. This warning can be removed using the argument "

'`attn\_implementation="eager"` when loading the model.'

)

return super().forward(

hidden\_states,

attention\_mask,

head\_mask,

encoder\_hidden\_states,

encoder\_attention\_mask,

past\_key\_value,

output\_attentions,

)

bsz, tgt\_len, \_ = hidden\_states.size()

query\_layer = self.transpose\_for\_scores(self.query(hidden\_states))

# If this is instantiated as a cross-attention module, the keys and values come from an encoder; the attention

# mask needs to be such that the encoder's padding tokens are not attended to.

is\_cross\_attention = encoder\_hidden\_states is not None

current\_states = encoder\_hidden\_states if is\_cross\_attention else hidden\_states

attention\_mask = encoder\_attention\_mask if is\_cross\_attention else attention\_mask

# Check `seq\_length` of `past\_key\_value` == `len(current\_states)` to support prefix tuning

if is\_cross\_attention and past\_key\_value and past\_key\_value[0].shape[2] == current\_states.shape[1]:

key\_layer, value\_layer = past\_key\_value

else:

key\_layer = self.transpose\_for\_scores(self.key(current\_states))

value\_layer = self.transpose\_for\_scores(self.value(current\_states))

if past\_key\_value is not None and not is\_cross\_attention:

key\_layer = torch.cat([past\_key\_value[0], key\_layer], dim=2)

value\_layer = torch.cat([past\_key\_value[1], value\_layer], dim=2)

if self.is\_decoder:

# if cross\_attention save Tuple(torch.Tensor, torch.Tensor) of all cross attention key/value\_states.

# Further calls to cross\_attention layer can then reuse all cross-attention

# key/value\_states (first "if" case)

# if uni-directional self-attention (decoder) save Tuple(torch.Tensor, torch.Tensor) of

# all previous decoder key/value\_states. Further calls to uni-directional self-attention

# can concat previous decoder key/value\_states to current projected key/value\_states (third "elif" case)

# if encoder bi-directional self-attention `past\_key\_value` is always `None`

past\_key\_value = (key\_layer, value\_layer)

# SDPA with memory-efficient backend is broken in torch==2.1.2 when using non-contiguous inputs and a custom

# attn\_mask, so we need to call `.contiguous()` here. This was fixed in torch==2.2.0.

# Reference: https://github.com/pytorch/pytorch/issues/112577

if self.require\_contiguous\_qkv and query\_layer.device.type == "cuda" and attention\_mask is not None:

query\_layer = query\_layer.contiguous()

key\_layer = key\_layer.contiguous()

value\_layer = value\_layer.contiguous()

# We dispatch to SDPA's Flash Attention or Efficient kernels via this `is\_causal` if statement instead of an inline conditional assignment

# in SDPA to support both torch.compile's dynamic shapes and full graph options. An inline conditional prevents dynamic shapes from compiling.

# The tgt\_len > 1 is necessary to match with AttentionMaskConverter.to\_causal\_4d that does not create

# a causal mask in case tgt\_len == 1.

is\_causal = (

True if self.is\_decoder and not is\_cross\_attention and attention\_mask is None and tgt\_len > 1 else False

)

attn\_output = torch.nn.functional.scaled\_dot\_product\_attention(

query\_layer,

key\_layer,

value\_layer,

attn\_mask=attention\_mask,

dropout\_p=self.dropout\_prob if self.training else 0.0,

is\_causal=is\_causal,

)

attn\_output = attn\_output.transpose(1, 2)

attn\_output = attn\_output.reshape(bsz, tgt\_len, self.all\_head\_size)

outputs = (attn\_output,)

if self.is\_decoder:

outputs = outputs + (past\_key\_value,)

return outputs

### Roberta Self Output

# Copied from transformers.models.bert.modeling\_bert.BertSelfOutput

class RobertaSelfOutput(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.dense = nn.Linear(config.hidden\_size, config.hidden\_size)

self.LayerNorm = nn.LayerNorm(config.hidden\_size, eps=config.layer\_norm\_eps)

self.dropout = nn.Dropout(config.hidden\_dropout\_prob)

def forward(self, hidden\_states: torch.Tensor, input\_tensor: torch.Tensor) -> torch.Tensor:

hidden\_states = self.dense(hidden\_states)

hidden\_states = self.dropout(hidden\_states)

hidden\_states = self.LayerNorm(hidden\_states + input\_tensor)

return hidden\_states

### Roberta Attention

# Copied from transformers.models.bert.modeling\_bert.BertAttention with Bert->Roberta,BERT->ROBERTA

class RobertaAttention(nn.Module):

def \_\_init\_\_(self, config, position\_embedding\_type=None):

super().\_\_init\_\_()

self.self = ROBERTA\_SELF\_ATTENTION\_CLASSES[config.\_attn\_implementation](

config, position\_embedding\_type=position\_embedding\_type

)

self.output = RobertaSelfOutput(config)

self.pruned\_heads = set()

def prune\_heads(self, heads):

if len(heads) == 0:

return

heads, index = find\_pruneable\_heads\_and\_indices(

heads, self.self.num\_attention\_heads, self.self.attention\_head\_size, self.pruned\_heads

)

# Prune linear layers

self.self.query = prune\_linear\_layer(self.self.query, index)

self.self.key = prune\_linear\_layer(self.self.key, index)

self.self.value = prune\_linear\_layer(self.self.value, index)

self.output.dense = prune\_linear\_layer(self.output.dense, index, dim=1)

# Update hyper params and store pruned heads

self.self.num\_attention\_heads = self.self.num\_attention\_heads - len(heads)

self.self.all\_head\_size = self.self.attention\_head\_size \* self.self.num\_attention\_heads

self.pruned\_heads = self.pruned\_heads.union(heads)

def forward(

self,

hidden\_states: torch.Tensor,

attention\_mask: Optional[torch.FloatTensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

encoder\_hidden\_states: Optional[torch.FloatTensor] = None,

encoder\_attention\_mask: Optional[torch.FloatTensor] = None,

past\_key\_value: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,

output\_attentions: Optional[bool] = False,

) -> Tuple[torch.Tensor]:

self\_outputs = self.self(

hidden\_states,

attention\_mask,

head\_mask,

encoder\_hidden\_states,

encoder\_attention\_mask,

past\_key\_value,

output\_attentions,

)

attention\_output = self.output(self\_outputs[0], hidden\_states)

outputs = (attention\_output,) + self\_outputs[1:] # add attentions if we output them

return outputs

### Roberta Intermediate

# Copied from transformers.models.bert.modeling\_bert.BertIntermediate

class RobertaIntermediate(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.dense = nn.Linear(config.hidden\_size, config.intermediate\_size)

if isinstance(config.hidden\_act, str):

self.intermediate\_act\_fn = ACT2FN[config.hidden\_act]

else:

self.intermediate\_act\_fn = config.hidden\_act

def forward(self, hidden\_states: torch.Tensor) -> torch.Tensor:

hidden\_states = self.dense(hidden\_states)

hidden\_states = self.intermediate\_act\_fn(hidden\_states)

return hidden\_states

### Roberta Output

# Copied from transformers.models.bert.modeling\_bert.BertOutput

class RobertaOutput(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.dense = nn.Linear(config.intermediate\_size, config.hidden\_size)

self.LayerNorm = nn.LayerNorm(config.hidden\_size, eps=config.layer\_norm\_eps)

self.dropout = nn.Dropout(config.hidden\_dropout\_prob)

def forward(self, hidden\_states: torch.Tensor, input\_tensor: torch.Tensor) -> torch.Tensor:

hidden\_states = self.dense(hidden\_states)

hidden\_states = self.dropout(hidden\_states)

hidden\_states = self.LayerNorm(hidden\_states + input\_tensor)

return hidden\_states

### Roberta Layer

# Copied from transformers.models.bert.modeling\_bert.BertLayer with Bert->Roberta

class RobertaLayer(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.chunk\_size\_feed\_forward = config.chunk\_size\_feed\_forward

self.seq\_len\_dim = 1

self.attention = RobertaAttention(config)

self.is\_decoder = config.is\_decoder

self.add\_cross\_attention = config.add\_cross\_attention

if self.add\_cross\_attention:

if not self.is\_decoder:

raise ValueError(f"{self} should be used as a decoder model if cross attention is added")

self.crossattention = RobertaAttention(config, position\_embedding\_type="absolute")

self.intermediate = RobertaIntermediate(config)

self.output = RobertaOutput(config)

def forward(

self,

hidden\_states: torch.Tensor,

attention\_mask: Optional[torch.FloatTensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

encoder\_hidden\_states: Optional[torch.FloatTensor] = None,

encoder\_attention\_mask: Optional[torch.FloatTensor] = None,

past\_key\_value: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,

output\_attentions: Optional[bool] = False,

) -> Tuple[torch.Tensor]:

# decoder uni-directional self-attention cached key/values tuple is at positions 1,2

self\_attn\_past\_key\_value = past\_key\_value[:2] if past\_key\_value is not None else None

self\_attention\_outputs = self.attention(

hidden\_states,

attention\_mask,

head\_mask,

output\_attentions=output\_attentions,

past\_key\_value=self\_attn\_past\_key\_value,

)

attention\_output = self\_attention\_outputs[0]

# if decoder, the last output is tuple of self-attn cache

if self.is\_decoder:

outputs = self\_attention\_outputs[1:-1]

present\_key\_value = self\_attention\_outputs[-1]

else:

outputs = self\_attention\_outputs[1:] # add self attentions if we output attention weights

cross\_attn\_present\_key\_value = None

if self.is\_decoder and encoder\_hidden\_states is not None:

if not hasattr(self, "crossattention"):

raise ValueError(

f"If `encoder\_hidden\_states` are passed, {self} has to be instantiated with cross-attention layers"

" by setting `config.add\_cross\_attention=True`"

)

# cross\_attn cached key/values tuple is at positions 3,4 of past\_key\_value tuple

cross\_attn\_past\_key\_value = past\_key\_value[-2:] if past\_key\_value is not None else None

cross\_attention\_outputs = self.crossattention(

attention\_output,

attention\_mask,

head\_mask,

encoder\_hidden\_states,

encoder\_attention\_mask,

cross\_attn\_past\_key\_value,

output\_attentions,

)

attention\_output = cross\_attention\_outputs[0]

outputs = outputs + cross\_attention\_outputs[1:-1] # add cross attentions if we output attention weights

# add cross-attn cache to positions 3,4 of present\_key\_value tuple

cross\_attn\_present\_key\_value = cross\_attention\_outputs[-1]

present\_key\_value = present\_key\_value + cross\_attn\_present\_key\_value

layer\_output = apply\_chunking\_to\_forward(

self.feed\_forward\_chunk, self.chunk\_size\_feed\_forward, self.seq\_len\_dim, attention\_output

)

outputs = (layer\_output,) + outputs

# if decoder, return the attn key/values as the last output

if self.is\_decoder:

outputs = outputs + (present\_key\_value,)

return outputs

def feed\_forward\_chunk(self, attention\_output):

intermediate\_output = self.intermediate(attention\_output)

layer\_output = self.output(intermediate\_output, attention\_output)

return layer\_output

### Roberta Encoder

# Copied from transformers.models.bert.modeling\_bert.BertEncoder with Bert->Roberta

class RobertaEncoder(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.config = config

self.layer = nn.ModuleList([RobertaLayer(config) for \_ in range(config.num\_hidden\_layers)])

self.gradient\_checkpointing = False

def forward(

self,

hidden\_states: torch.Tensor,

attention\_mask: Optional[torch.FloatTensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

encoder\_hidden\_states: Optional[torch.FloatTensor] = None,

encoder\_attention\_mask: Optional[torch.FloatTensor] = None,

past\_key\_values: Optional[Tuple[Tuple[torch.FloatTensor]]] = None,

use\_cache: Optional[bool] = None,

output\_attentions: Optional[bool] = False,

output\_hidden\_states: Optional[bool] = False,

return\_dict: Optional[bool] = True,

) -> Union[Tuple[torch.Tensor], BaseModelOutputWithPastAndCrossAttentions]:

all\_hidden\_states = () if output\_hidden\_states else None

all\_self\_attentions = () if output\_attentions else None

all\_cross\_attentions = () if output\_attentions and self.config.add\_cross\_attention else None

if self.gradient\_checkpointing and self.training:

if use\_cache:

logger.warning\_once(

"`use\_cache=True` is incompatible with gradient checkpointing. Setting `use\_cache=False`..."

)

use\_cache = False

next\_decoder\_cache = () if use\_cache else None

for i, layer\_module in enumerate(self.layer):

if output\_hidden\_states:

all\_hidden\_states = all\_hidden\_states + (hidden\_states,)

layer\_head\_mask = head\_mask[i] if head\_mask is not None else None

past\_key\_value = past\_key\_values[i] if past\_key\_values is not None else None

if self.gradient\_checkpointing and self.training:

layer\_outputs = self.\_gradient\_checkpointing\_func(

layer\_module.\_\_call\_\_,

hidden\_states,

attention\_mask,

layer\_head\_mask,

encoder\_hidden\_states,

encoder\_attention\_mask,

past\_key\_value,

output\_attentions,

)

else:

layer\_outputs = layer\_module(

hidden\_states,

attention\_mask,

layer\_head\_mask,

encoder\_hidden\_states,

encoder\_attention\_mask,

past\_key\_value,

output\_attentions,

)

hidden\_states = layer\_outputs[0]

if use\_cache:

next\_decoder\_cache += (layer\_outputs[-1],)

if output\_attentions:

all\_self\_attentions = all\_self\_attentions + (layer\_outputs[1],)

if self.config.add\_cross\_attention:

all\_cross\_attentions = all\_cross\_attentions + (layer\_outputs[2],)

if output\_hidden\_states:

all\_hidden\_states = all\_hidden\_states + (hidden\_states,)

if not return\_dict:

return tuple(

v

for v in [

hidden\_states,

next\_decoder\_cache,

all\_hidden\_states,

all\_self\_attentions,

all\_cross\_attentions,

]

if v is not None

)

return BaseModelOutputWithPastAndCrossAttentions(

last\_hidden\_state=hidden\_states,

past\_key\_values=next\_decoder\_cache,

hidden\_states=all\_hidden\_states,

attentions=all\_self\_attentions,

cross\_attentions=all\_cross\_attentions,

)

### Roberta Pooler

# Copied from transformers.models.bert.modeling\_bert.BertPooler

class RobertaPooler(nn.Module):

def \_\_init\_\_(self, config):

super().\_\_init\_\_()

self.dense = nn.Linear(config.hidden\_size, config.hidden\_size)

self.activation = nn.Tanh()

def forward(self, hidden\_states: torch.Tensor) -> torch.Tensor:

# We "pool" the model by simply taking the hidden state corresponding

# to the first token.

first\_token\_tensor = hidden\_states[:, 0]

pooled\_output = self.dense(first\_token\_tensor)

pooled\_output = self.activation(pooled\_output)

return pooled\_output

### Base pretrained Roberta Model

class RobertaPreTrainedModel(PreTrainedModel):

"""

An abstract class to handle weights initialization and a simple interface for downloading and loading pretrained

models.

"""

config\_class = RobertaConfig

base\_model\_prefix = "roberta"

supports\_gradient\_checkpointing = True

\_no\_split\_modules = ["RobertaEmbeddings", "RobertaSelfAttention", "RobertaSdpaSelfAttention"]

\_supports\_sdpa = True

# Copied from transformers.models.bert.modeling\_bert.BertPreTrainedModel.\_init\_weights with BertLMPredictionHead->RobertaLMHead

def \_init\_weights(self, module):

"""Initialize the weights"""

if isinstance(module, nn.Linear):

# Slightly different from the TF version which uses truncated\_normal for initialization

# cf https://github.com/pytorch/pytorch/pull/5617

module.weight.data.normal\_(mean=0.0, std=self.config.initializer\_range)

if module.bias is not None:

module.bias.data.zero\_()

elif isinstance(module, nn.Embedding):

module.weight.data.normal\_(mean=0.0, std=self.config.initializer\_range)

if module.padding\_idx is not None:

module.weight.data[module.padding\_idx].zero\_()

elif isinstance(module, nn.LayerNorm):

module.bias.data.zero\_()

module.weight.data.fill\_(1.0)

elif isinstance(module, RobertaLMHead):

module.bias.data.zero\_()

### General Purpose Roberta Model

**Description:**

This model inherits from PreTrainedModel. Check the superclass documentation for the generic methods the

library implements for all its model (such as downloading or saving, resizing the input embeddings, pruning heads

etc.)

This model is also a PyTorch [torch.nn.Module](https://pytorch.org/docs/stable/nn.html#torch.nn.Module) subclass.

Use it as a regular PyTorch Module and refer to the PyTorch documentation for all matter related to general usage

and behavior.

Parameters:

config (RobertaConfig): Model configuration class with all the parameters of the

model. Initializing with a config file does not load the weights associated with the model, only the

configuration. Check out the ~PreTrainedModel.from\_pretrained method to load the model weights.

**Doc string:**

Args:

input\_ids (`torch.LongTensor` of shape `({0})`):

Indices of input sequence tokens in the vocabulary.

Indices can be obtained using [`AutoTokenizer`]. See [`PreTrainedTokenizer.encode`] and

[`PreTrainedTokenizer.\_\_call\_\_`] for details.

[What are input IDs?](../glossary#input-ids)

attention\_mask (`torch.FloatTensor` of shape `({0})`, \*optional\*):

Mask to avoid performing attention on padding token indices. Mask values selected in `[0, 1]`:

- 1 for tokens that are \*\*not masked\*\*,

- 0 for tokens that are \*\*masked\*\*.

[What are attention masks?](../glossary#attention-mask)

token\_type\_ids (`torch.LongTensor` of shape `({0})`, \*optional\*):

Segment token indices to indicate first and second portions of the inputs. Indices are selected in `[0,1]`:

- 0 corresponds to a \*sentence A\* token,

- 1 corresponds to a \*sentence B\* token.

This parameter can only be used when the model is initialized with `type\_vocab\_size` parameter with value

>= 2. All the value in this tensor should be always < type\_vocab\_size.

[What are token type IDs?](../glossary#token-type-ids)

position\_ids (`torch.LongTensor` of shape `({0})`, \*optional\*):

Indices of positions of each input sequence tokens in the position embeddings. Selected in the range `[0,

config.max\_position\_embeddings - 1]`.

[What are position IDs?](../glossary#position-ids)

head\_mask (`torch.FloatTensor` of shape `(num\_heads,)` or `(num\_layers, num\_heads)`, \*optional\*):

Mask to nullify selected heads of the self-attention modules. Mask values selected in `[0, 1]`:

- 1 indicates the head is \*\*not masked\*\*,

- 0 indicates the head is \*\*masked\*\*.

inputs\_embeds (`torch.FloatTensor` of shape `({0}, hidden\_size)`, \*optional\*):

Optionally, instead of passing `input\_ids` you can choose to directly pass an embedded representation. This

is useful if you want more control over how to convert `input\_ids` indices into associated vectors than the

model's internal embedding lookup matrix.

output\_attentions (`bool`, \*optional\*):

Whether or not to return the attentions tensors of all attention layers. See `attentions` under returned

tensors for more detail.

output\_hidden\_states (`bool`, \*optional\*):

Whether or not to return the hidden states of all layers. See `hidden\_states` under returned tensors for

more detail.

return\_dict (`bool`, \*optional\*):

Whether or not to return a [`~utils.ModelOutput`] instead of a plain tuple.

# Copied from transformers.models.bert.modeling\_bert.BertModel with Bert->Roberta, BERT->ROBERTA

class RobertaModel(RobertaPreTrainedModel):

"""

The model can behave as an encoder (with only self-attention) as well as a decoder, in which case a layer of

cross-attention is added between the self-attention layers, following the architecture described in [Attention is

all you need](https://arxiv.org/abs/1706.03762) by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit,

Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin.

To behave as an decoder the model needs to be initialized with the `is\_decoder` argument of the configuration set

to `True`. To be used in a Seq2Seq model, the model needs to initialized with both `is\_decoder` argument and

`add\_cross\_attention` set to `True`; an `encoder\_hidden\_states` is then expected as an input to the forward pass.

"""

\_no\_split\_modules = ["RobertaEmbeddings", "RobertaLayer"]

def \_\_init\_\_(self, config, add\_pooling\_layer=True):

super().\_\_init\_\_(config)

self.config = config

self.embeddings = RobertaEmbeddings(config)

self.encoder = RobertaEncoder(config)

self.pooler = RobertaPooler(config) if add\_pooling\_layer else None

self.attn\_implementation = config.\_attn\_implementation

self.position\_embedding\_type = config.position\_embedding\_type

# Initialize weights and apply final processing

self.post\_init()

def get\_input\_embeddings(self):

return self.embeddings.word\_embeddings

def set\_input\_embeddings(self, value):

self.embeddings.word\_embeddings = value

def \_prune\_heads(self, heads\_to\_prune):

"""

Prunes heads of the model. heads\_to\_prune: dict of {layer\_num: list of heads to prune in this layer} See base

class PreTrainedModel

"""

for layer, heads in heads\_to\_prune.items():

self.encoder.layer[layer].attention.prune\_heads(heads)

@add\_start\_docstrings\_to\_model\_forward(ROBERTA\_INPUTS\_DOCSTRING.format("batch\_size, sequence\_length"))

@add\_code\_sample\_docstrings(

checkpoint=\_CHECKPOINT\_FOR\_DOC,

output\_type=BaseModelOutputWithPoolingAndCrossAttentions,

config\_class=\_CONFIG\_FOR\_DOC,

)

def forward(

self,

input\_ids: Optional[torch.Tensor] = None,

attention\_mask: Optional[torch.Tensor] = None,

token\_type\_ids: Optional[torch.Tensor] = None,

position\_ids: Optional[torch.Tensor] = None,

head\_mask: Optional[torch.Tensor] = None,

inputs\_embeds: Optional[torch.Tensor] = None,

encoder\_hidden\_states: Optional[torch.Tensor] = None,

encoder\_attention\_mask: Optional[torch.Tensor] = None,

past\_key\_values: Optional[List[torch.FloatTensor]] = None,

use\_cache: Optional[bool] = None,

output\_attentions: Optional[bool] = None,

output\_hidden\_states: Optional[bool] = None,

return\_dict: Optional[bool] = None,

) -> Union[Tuple[torch.Tensor], BaseModelOutputWithPoolingAndCrossAttentions]:

r"""

encoder\_hidden\_states (`torch.FloatTensor` of shape `(batch\_size, sequence\_length, hidden\_size)`, \*optional\*):

Sequence of hidden-states at the output of the last layer of the encoder. Used in the cross-attention if

the model is configured as a decoder.

encoder\_attention\_mask (`torch.FloatTensor` of shape `(batch\_size, sequence\_length)` or `(batch\_size, sequence\_length, target\_length)`, \*optional\*):

Mask to avoid performing attention on the padding token indices of the encoder input. This mask is used in

the cross-attention if the model is configured as a decoder. Mask values selected in `[0, 1]`:

- 1 for tokens that are \*\*not masked\*\*,

- 0 for tokens that are \*\*masked\*\*.

past\_key\_values (`tuple(tuple(torch.FloatTensor))` of length `config.n\_layers` with each tuple having 4 tensors of shape `(batch\_size, num\_heads, sequence\_length - 1, embed\_size\_per\_head)`):

Contains precomputed key and value hidden states of the attention blocks. Can be used to speed up decoding.

If `past\_key\_values` are used, the user can optionally input only the last `decoder\_input\_ids` (those that

don't have their past key value states given to this model) of shape `(batch\_size, 1)` instead of all

`decoder\_input\_ids` of shape `(batch\_size, sequence\_length)`.

use\_cache (`bool`, \*optional\*):

If set to `True`, `past\_key\_values` key value states are returned and can be used to speed up decoding (see

`past\_key\_values`).

"""

output\_attentions = output\_attentions if output\_attentions is not None else self.config.output\_attentions

output\_hidden\_states = (

output\_hidden\_states if output\_hidden\_states is not None else self.config.output\_hidden\_states

)

return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict

if self.config.is\_decoder:

use\_cache = use\_cache if use\_cache is not None else self.config.use\_cache

else:

use\_cache = False

if input\_ids is not None and inputs\_embeds is not None:

raise ValueError("You cannot specify both input\_ids and inputs\_embeds at the same time")

elif input\_ids is not None:

self.warn\_if\_padding\_and\_no\_attention\_mask(input\_ids, attention\_mask)

input\_shape = input\_ids.size()

elif inputs\_embeds is not None:

input\_shape = inputs\_embeds.size()[:-1]

else:

raise ValueError("You have to specify either input\_ids or inputs\_embeds")

batch\_size, seq\_length = input\_shape

device = input\_ids.device if input\_ids is not None else inputs\_embeds.device

# past\_key\_values\_length

past\_key\_values\_length = past\_key\_values[0][0].shape[2] if past\_key\_values is not None else 0

if token\_type\_ids is None:

if hasattr(self.embeddings, "token\_type\_ids"):

buffered\_token\_type\_ids = self.embeddings.token\_type\_ids[:, :seq\_length]

buffered\_token\_type\_ids\_expanded = buffered\_token\_type\_ids.expand(batch\_size, seq\_length)

token\_type\_ids = buffered\_token\_type\_ids\_expanded

else:

token\_type\_ids = torch.zeros(input\_shape, dtype=torch.long, device=device)

embedding\_output = self.embeddings(

input\_ids=input\_ids,

position\_ids=position\_ids,

token\_type\_ids=token\_type\_ids,

inputs\_embeds=inputs\_embeds,

past\_key\_values\_length=past\_key\_values\_length,

)

if attention\_mask is None:

attention\_mask = torch.ones((batch\_size, seq\_length + past\_key\_values\_length), device=device)

use\_sdpa\_attention\_masks = (

self.attn\_implementation == "sdpa"

and self.position\_embedding\_type == "absolute"

and head\_mask is None

and not output\_attentions

)

# Expand the attention mask

if use\_sdpa\_attention\_masks and attention\_mask.dim() == 2:

# Expand the attention mask for SDPA.

# [bsz, seq\_len] -> [bsz, 1, seq\_len, seq\_len]

if self.config.is\_decoder:

extended\_attention\_mask = \_prepare\_4d\_causal\_attention\_mask\_for\_sdpa(

attention\_mask,

input\_shape,

embedding\_output,

past\_key\_values\_length,

)

else:

extended\_attention\_mask = \_prepare\_4d\_attention\_mask\_for\_sdpa(

attention\_mask, embedding\_output.dtype, tgt\_len=seq\_length

)

else:

# We can provide a self-attention mask of dimensions [batch\_size, from\_seq\_length, to\_seq\_length]

# ourselves in which case we just need to make it broadcastable to all heads.

extended\_attention\_mask = self.get\_extended\_attention\_mask(attention\_mask, input\_shape)

# If a 2D or 3D attention mask is provided for the cross-attention

# we need to make broadcastable to [batch\_size, num\_heads, seq\_length, seq\_length]

if self.config.is\_decoder and encoder\_hidden\_states is not None:

encoder\_batch\_size, encoder\_sequence\_length, \_ = encoder\_hidden\_states.size()

encoder\_hidden\_shape = (encoder\_batch\_size, encoder\_sequence\_length)

if encoder\_attention\_mask is None:

encoder\_attention\_mask = torch.ones(encoder\_hidden\_shape, device=device)

if use\_sdpa\_attention\_masks and encoder\_attention\_mask.dim() == 2:

# Expand the attention mask for SDPA.

# [bsz, seq\_len] -> [bsz, 1, seq\_len, seq\_len]

encoder\_extended\_attention\_mask = \_prepare\_4d\_attention\_mask\_for\_sdpa(

encoder\_attention\_mask, embedding\_output.dtype, tgt\_len=seq\_length

)

else:

encoder\_extended\_attention\_mask = self.invert\_attention\_mask(encoder\_attention\_mask)

else:

encoder\_extended\_attention\_mask = None

# Prepare head mask if needed

# 1.0 in head\_mask indicate we keep the head

# attention\_probs has shape bsz x n\_heads x N x N

# input head\_mask has shape [num\_heads] or [num\_hidden\_layers x num\_heads]

# and head\_mask is converted to shape [num\_hidden\_layers x batch x num\_heads x seq\_length x seq\_length]

head\_mask = self.get\_head\_mask(head\_mask, self.config.num\_hidden\_layers)

encoder\_outputs = self.encoder(

embedding\_output,

attention\_mask=extended\_attention\_mask,

head\_mask=head\_mask,

encoder\_hidden\_states=encoder\_hidden\_states,

encoder\_attention\_mask=encoder\_extended\_attention\_mask,

past\_key\_values=past\_key\_values,

use\_cache=use\_cache,

output\_attentions=output\_attentions,

output\_hidden\_states=output\_hidden\_states,

return\_dict=return\_dict,

)

sequence\_output = encoder\_outputs[0]

pooled\_output = self.pooler(sequence\_output) if self.pooler is not None else None

if not return\_dict:

return (sequence\_output, pooled\_output) + encoder\_outputs[1:]

return BaseModelOutputWithPoolingAndCrossAttentions(

last\_hidden\_state=sequence\_output,

pooler\_output=pooled\_output,

past\_key\_values=encoder\_outputs.past\_key\_values,

hidden\_states=encoder\_outputs.hidden\_states,

attentions=encoder\_outputs.attentions,

cross\_attentions=encoder\_outputs.cross\_attentions,

)

### Roberta Model for Sequence Classification

**Description:**

RoBERTa Model transformer with a sequence classification/regression head on top (a linear layer on top of the

pooled output) e.g. for GLUE tasks.

class RobertaForSequenceClassification(RobertaPreTrainedModel):

def \_\_init\_\_(self, config):

super().\_\_init\_\_(config)

self.num\_labels = config.num\_labels

self.config = config

self.roberta = RobertaModel(config, add\_pooling\_layer=False)

self.classifier = RobertaClassificationHead(config)

# Initialize weights and apply final processing

self.post\_init()

@add\_start\_docstrings\_to\_model\_forward(ROBERTA\_INPUTS\_DOCSTRING.format("batch\_size, sequence\_length"))

@add\_code\_sample\_docstrings(

checkpoint="cardiffnlp/twitter-roberta-base-emotion",

output\_type=SequenceClassifierOutput,

config\_class=\_CONFIG\_FOR\_DOC,

expected\_output="'optimism'",

expected\_loss=0.08,

)

def forward(

self,

input\_ids: Optional[torch.LongTensor] = None,

attention\_mask: Optional[torch.FloatTensor] = None,

token\_type\_ids: Optional[torch.LongTensor] = None,

position\_ids: Optional[torch.LongTensor] = None,

head\_mask: Optional[torch.FloatTensor] = None,

inputs\_embeds: Optional[torch.FloatTensor] = None,

labels: Optional[torch.LongTensor] = None,

output\_attentions: Optional[bool] = None,

output\_hidden\_states: Optional[bool] = None,

return\_dict: Optional[bool] = None,

) -> Union[Tuple[torch.Tensor], SequenceClassifierOutput]:

r"""

labels (`torch.LongTensor` of shape `(batch\_size,)`, \*optional\*):

Labels for computing the sequence classification/regression loss. Indices should be in `[0, ...,

config.num\_labels - 1]`. If `config.num\_labels == 1` a regression loss is computed (Mean-Square loss), If

`config.num\_labels > 1` a classification loss is computed (Cross-Entropy).

"""

return\_dict = return\_dict if return\_dict is not None else self.config.use\_return\_dict

outputs = self.roberta(

input\_ids,

attention\_mask=attention\_mask,

token\_type\_ids=token\_type\_ids,

position\_ids=position\_ids,

head\_mask=head\_mask,

inputs\_embeds=inputs\_embeds,

output\_attentions=output\_attentions,

output\_hidden\_states=output\_hidden\_states,

return\_dict=return\_dict,

)

sequence\_output = outputs[0]

logits = self.classifier(sequence\_output)

loss = None

if labels is not None:

# move labels to correct device to enable model parallelism

labels = labels.to(logits.device)

if self.config.problem\_type is None:

if self.num\_labels == 1:

self.config.problem\_type = "regression"

elif self.num\_labels > 1 and (labels.dtype == torch.long or labels.dtype == torch.int):

self.config.problem\_type = "single\_label\_classification"

else:

self.config.problem\_type = "multi\_label\_classification"

if self.config.problem\_type == "regression":

loss\_fct = MSELoss()

if self.num\_labels == 1:

loss = loss\_fct(logits.squeeze(), labels.squeeze())

else:

loss = loss\_fct(logits, labels)

elif self.config.problem\_type == "single\_label\_classification":

loss\_fct = CrossEntropyLoss()

loss = loss\_fct(logits.view(-1, self.num\_labels), labels.view(-1))

elif self.config.problem\_type == "multi\_label\_classification":

loss\_fct = BCEWithLogitsLoss()

loss = loss\_fct(logits, labels)

if not return\_dict:

output = (logits,) + outputs[2:]

return ((loss,) + output) if loss is not None else output

return SequenceClassifierOutput(

loss=loss,

logits=logits,

hidden\_states=outputs.hidden\_states,

attentions=outputs.attentions,

)

## Insider the HF Transformers Trainer module

<https://github.com/huggingface/transformers/blob/main/src/transformers/trainer.py>

### The Trainer class

class Trainer:

"""

Trainer is a simple but feature-complete training and eval loop for PyTorch, optimized for 🤗 Transformers.

Args:

model ([`PreTrainedModel`] or `torch.nn.Module`, \*optional\*):

The model to train, evaluate or use for predictions. If not provided, a `model\_init` must be passed.

<Tip>

[`Trainer`] is optimized to work with the [`PreTrainedModel`] provided by the library. You can still use

your own models defined as `torch.nn.Module` as long as they work the same way as the 🤗 Transformers

models.

</Tip>

args ([`TrainingArguments`], \*optional\*):

The arguments to tweak for training. Will default to a basic instance of [`TrainingArguments`] with the

`output\_dir` set to a directory named \*tmp\_trainer\* in the current directory if not provided.

data\_collator (`DataCollator`, \*optional\*):

The function to use to form a batch from a list of elements of `train\_dataset` or `eval\_dataset`. Will

default to [`default\_data\_collator`] if no `processing\_class` is provided, an instance of

[`DataCollatorWithPadding`] otherwise if the processing\_class is a feature extractor or tokenizer.

train\_dataset (Union[`torch.utils.data.Dataset`, `torch.utils.data.IterableDataset`, `datasets.Dataset`], \*optional\*):

The dataset to use for training. If it is a [`~datasets.Dataset`], columns not accepted by the

`model.forward()` method are automatically removed.

Note that if it's a `torch.utils.data.IterableDataset` with some randomization and you are training in a

distributed fashion, your iterable dataset should either use a internal attribute `generator` that is a

`torch.Generator` for the randomization that must be identical on all processes (and the Trainer will

manually set the seed of this `generator` at each epoch) or have a `set\_epoch()` method that internally

sets the seed of the RNGs used.

eval\_dataset (Union[`torch.utils.data.Dataset`, Dict[str, `torch.utils.data.Dataset`, `datasets.Dataset`]), \*optional\*):

The dataset to use for evaluation. If it is a [`~datasets.Dataset`], columns not accepted by the

`model.forward()` method are automatically removed. If it is a dictionary, it will evaluate on each

dataset prepending the dictionary key to the metric name.

processing\_class (`PreTrainedTokenizerBase` or `BaseImageProcessor` or `FeatureExtractionMixin` or `ProcessorMixin`, \*optional\*):

Processing class used to process the data. If provided, will be used to automatically process the inputs

for the model, and it will be saved along the model to make it easier to rerun an interrupted training or

reuse the fine-tuned model.

This supercedes the `tokenizer` argument, which is now deprecated.

model\_init (`Callable[[], PreTrainedModel]`, \*optional\*):

A function that instantiates the model to be used. If provided, each call to [`~Trainer.train`] will start

from a new instance of the model as given by this function.

The function may have zero argument, or a single one containing the optuna/Ray Tune/SigOpt trial object, to

be able to choose different architectures according to hyper parameters (such as layer count, sizes of

inner layers, dropout probabilities etc).

compute\_loss\_func (`Callable`, \*optional\*):

A function that accepts the raw model outputs, labels, and the number of items in the entire accumulated

batch (batch\_size \* gradient\_accumulation\_steps) and returns the loss. For example, see the default [loss function](https://github.com/huggingface/transformers/blob/052e652d6d53c2b26ffde87e039b723949a53493/src/transformers/trainer.py#L3618) used by [`Trainer`].

compute\_metrics (`Callable[[EvalPrediction], Dict]`, \*optional\*):

The function that will be used to compute metrics at evaluation. Must take a [`EvalPrediction`] and return

a dictionary string to metric values. \*Note\* When passing TrainingArgs with `batch\_eval\_metrics` set to

`True`, your compute\_metrics function must take a boolean `compute\_result` argument. This will be triggered

after the last eval batch to signal that the function needs to calculate and return the global summary

statistics rather than accumulating the batch-level statistics

callbacks (List of [`TrainerCallback`], \*optional\*):

A list of callbacks to customize the training loop. Will add those to the list of default callbacks

detailed in [here](callback).

If you want to remove one of the default callbacks used, use the [`Trainer.remove\_callback`] method.

optimizers (`Tuple[torch.optim.Optimizer, torch.optim.lr\_scheduler.LambdaLR]`, \*optional\*, defaults to `(None, None)`):

A tuple containing the optimizer and the scheduler to use. Will default to an instance of [`AdamW`] on your

model and a scheduler given by [`get\_linear\_schedule\_with\_warmup`] controlled by `args`.

optimizer\_cls\_and\_kwargs (`Tuple[Type[torch.optim.Optimizer], Dict[str, Any]]`, \*optional\*):

A tuple containing the optimizer class and keyword arguments to use.

Overrides `optim` and `optim\_args` in `args`. Incompatible with the `optimizers` argument.

Unlike `optimizers`, this argument avoids the need to place model parameters on the correct devices before initializing the Trainer.

preprocess\_logits\_for\_metrics (`Callable[[torch.Tensor, torch.Tensor], torch.Tensor]`, \*optional\*):

A function that preprocess the logits right before caching them at each evaluation step. Must take two

tensors, the logits, and the labels, and return the logits once processed as desired. The modifications made

by this function will be reflected in the predictions received by `compute\_metrics`.

Note that the labels (second parameter) will be `None` if the dataset does not have them.

Important attributes:

- \*\*model\*\* -- Always points to the core model. If using a transformers model, it will be a [`PreTrainedModel`]

subclass.

- \*\*model\_wrapped\*\* -- Always points to the most external model in case one or more other modules wrap the

original model. This is the model that should be used for the forward pass. For example, under `DeepSpeed`,

the inner model is wrapped in `DeepSpeed` and then again in `torch.nn.DistributedDataParallel`. If the inner

model hasn't been wrapped, then `self.model\_wrapped` is the same as `self.model`.

- \*\*is\_model\_parallel\*\* -- Whether or not a model has been switched to a model parallel mode (different from

data parallelism, this means some of the model layers are split on different GPUs).

- \*\*place\_model\_on\_device\*\* -- Whether or not to automatically place the model on the device - it will be set

to `False` if model parallel or deepspeed is used, or if the default

`TrainingArguments.place\_model\_on\_device` is overridden to return `False` .

- \*\*is\_in\_train\*\* -- Whether or not a model is currently running `train` (e.g. when `evaluate` is called while

in `train`)

"""

# Those are used as methods of the Trainer in examples.

from .trainer\_pt\_utils import \_get\_learning\_rate, log\_metrics, metrics\_format, save\_metrics, save\_state

@deprecate\_kwarg("tokenizer", new\_name="processing\_class", version="5.0.0", raise\_if\_both\_names=True)

def \_\_init\_\_(

self,

model: Union[PreTrainedModel, nn.Module, None] = None,

args: TrainingArguments = None,

data\_collator: Optional[DataCollator] = None,

train\_dataset: Optional[Union[Dataset, IterableDataset, "datasets.Dataset"]] = None,

eval\_dataset: Optional[Union[Dataset, dict[str, Dataset], "datasets.Dataset"]] = None,

processing\_class: Optional[

Union[PreTrainedTokenizerBase, BaseImageProcessor, FeatureExtractionMixin, ProcessorMixin]

] = None,

model\_init: Optional[Callable[[], PreTrainedModel]] = None,

compute\_loss\_func: Optional[Callable] = None,

compute\_metrics: Optional[Callable[[EvalPrediction], dict]] = None,

callbacks: Optional[list[TrainerCallback]] = None,

optimizers: tuple[Optional[torch.optim.Optimizer], Optional[torch.optim.lr\_scheduler.LambdaLR]] = (None, None),

optimizer\_cls\_and\_kwargs: Optional[tuple[type[torch.optim.Optimizer], dict[str, Any]]] = None,

preprocess\_logits\_for\_metrics: Optional[Callable[[torch.Tensor, torch.Tensor], torch.Tensor]] = None,

):

if args is None:

output\_dir = "tmp\_trainer"

logger.info(f"No `TrainingArguments` passed, using `output\_dir={output\_dir}`.")

args = TrainingArguments(output\_dir=output\_dir)

if args.batch\_eval\_metrics and compute\_metrics is not None:

if "compute\_result" not in inspect.signature(compute\_metrics).parameters.keys():

raise ValueError(

"When using `batch\_eval\_metrics`, your `compute\_metrics` function must take a `compute\_result`"

" boolean argument which will be triggered after the last batch of the eval set to signal that the"

" summary statistics should be returned by the function."

)

if args.eval\_strategy is not None and args.eval\_strategy != "no" and eval\_dataset is None:

raise ValueError(

f"You have set `args.eval\_strategy` to {args.eval\_strategy} but you didn't pass an `eval\_dataset` to `Trainer`. Either set `args.eval\_strategy` to `no` or pass an `eval\_dataset`. "

)

if args.save\_strategy == SaveStrategy.BEST or args.load\_best\_model\_at\_end:

if args.metric\_for\_best\_model is None:

raise ValueError(

"`args.metric\_for\_best\_model` must be provided when using 'best' save\_strategy or if `args.load\_best\_model\_at\_end` is set to `True`."

)

self.args = args

self.compute\_loss\_func = compute\_loss\_func

# Seed must be set before instantiating the model when using model

enable\_full\_determinism(self.args.seed) if self.args.full\_determinism else set\_seed(self.args.seed)

self.hp\_name = None

self.deepspeed = None

self.is\_in\_train = False

self.model = model

self.create\_accelerator\_and\_postprocess()

# memory metrics - must set up as early as possible

self.\_memory\_tracker = TrainerMemoryTracker(self.args.skip\_memory\_metrics)

self.\_memory\_tracker.start()

# set the correct log level depending on the node

log\_level = args.get\_process\_log\_level()

logging.set\_verbosity(log\_level)

# force device and distributed setup init explicitly

args.\_setup\_devices

if model is None:

if model\_init is not None:

self.model\_init = model\_init

model = self.call\_model\_init()

else:

raise RuntimeError("`Trainer` requires either a `model` or `model\_init` argument")

else:

if model\_init is not None:

warnings.warn(

"`Trainer` requires either a `model` or `model\_init` argument, but not both. `model\_init` will"

" overwrite your model when calling the `train` method. This will become a fatal error in the next"

" release.",

FutureWarning,

)

self.model\_init = model\_init

if model.\_\_class\_\_.\_\_name\_\_ in MODEL\_MAPPING\_NAMES:

raise ValueError(

f"The model you have picked ({model.\_\_class\_\_.\_\_name\_\_}) cannot be used as is for training: it only "

"computes hidden states and does not accept any labels. You should choose a model with a head "

"suitable for your task like any of the `AutoModelForXxx` listed at "

"https://huggingface.co/docs/transformers/model\_doc/auto"

)

if getattr(model, "is\_parallelizable", False) and getattr(model, "model\_parallel", False):

self.is\_model\_parallel = True

else:

self.is\_model\_parallel = False

if getattr(model, "hf\_device\_map", None) is not None:

devices = [device for device in set(model.hf\_device\_map.values()) if device not in ["cpu", "disk"]]

if len(devices) > 1:

self.is\_model\_parallel = True

elif len(devices) == 1:

self.is\_model\_parallel = self.args.device != torch.device(devices[0])

else:

self.is\_model\_parallel = False

# warn users

if self.is\_model\_parallel:

logger.info(

"You have loaded a model on multiple GPUs. `is\_model\_parallel` attribute will be force-set"

" to `True` to avoid any unexpected behavior such as device placement mismatching."

)

if self.args.use\_liger\_kernel:

if is\_liger\_kernel\_available():

from liger\_kernel.transformers import \_apply\_liger\_kernel\_to\_instance

if isinstance(model, PreTrainedModel):

# Patch the model with liger kernels. Use the default kernel configurations.

\_apply\_liger\_kernel\_to\_instance(model=model)

elif hasattr(model, "get\_base\_model") and isinstance(model.get\_base\_model(), PreTrainedModel):

# Patch the base model with liger kernels where model is a PeftModel. Use the default kernel configurations.

\_apply\_liger\_kernel\_to\_instance(model=model.get\_base\_model())

else:

logger.warning(

"The model is not an instance of PreTrainedModel. No liger kernels will be applied."

)

else:

raise ImportError(

"You have set `use\_liger\_kernel` to `True` but liger-kernel >= 0.3.0 is not available. "

"Please install it with `pip install liger-kernel`"

)

\_is\_quantized\_and\_base\_model = getattr(model, "is\_quantized", False) and not getattr(

model, "\_hf\_peft\_config\_loaded", False

)

\_quantization\_method\_supports\_training = (

getattr(model, "hf\_quantizer", None) is not None and model.hf\_quantizer.is\_trainable

)

\_is\_model\_quantized\_and\_qat\_trainable = getattr(model, "hf\_quantizer", None) is not None and getattr(

model.hf\_quantizer, "is\_qat\_trainable", False

)

# Filter out quantized + compiled models

if \_is\_quantized\_and\_base\_model and hasattr(model, "\_orig\_mod"):

raise ValueError(

"You cannot fine-tune quantized model with `torch.compile()` make sure to pass a non-compiled model when fine-tuning a quantized model with PEFT"

)

# At this stage the model is already loaded

if \_is\_quantized\_and\_base\_model and not \_is\_peft\_model(model) and not \_is\_model\_quantized\_and\_qat\_trainable:

raise ValueError(

"You cannot perform fine-tuning on purely quantized models. Please attach trainable adapters on top of"

" the quantized model to correctly perform fine-tuning. Please see: https://huggingface.co/docs/transformers/peft"

" for more details"

)

elif \_is\_quantized\_and\_base\_model and not \_quantization\_method\_supports\_training:

raise ValueError(

f"The model you are trying to fine-tune is quantized with {model.hf\_quantizer.quantization\_config.quant\_method}"

" but that quantization method do not support training. Please open an issue on GitHub: https://github.com/huggingface/transformers"

f" to request the support for training support for {model.hf\_quantizer.quantization\_config.quant\_method}"

)

self.is\_fsdp\_xla\_enabled = args.fsdp\_config["xla"]

if len(args.fsdp) > 0:

if self.is\_deepspeed\_enabled:

raise ValueError(

"Using --fsdp xxx together with --deepspeed is not possible, deactivate one of those flags."

)

if not args.fsdp\_config["xla"] and args.parallel\_mode != ParallelMode.DISTRIBUTED:

raise ValueError("Using fsdp only works in distributed training.")

# one place to sort out whether to place the model on device or not

# postpone switching model to cuda when:

# 1. MP - since we are trying to fit a much bigger than 1 gpu model

# 2. fp16-enabled DeepSpeed loads the model in half the size and it doesn't need .to() anyway,

# and we only use deepspeed for training at the moment

# 3. full bf16 or fp16 eval - since the model needs to be cast to the right dtype first

# 4. FSDP - same as MP

self.place\_model\_on\_device = args.place\_model\_on\_device

if (

self.is\_model\_parallel

or self.is\_deepspeed\_enabled

or ((args.fp16\_full\_eval or args.bf16\_full\_eval) and not args.do\_train)

or self.is\_fsdp\_xla\_enabled

or self.is\_fsdp\_enabled

):

self.place\_model\_on\_device = False

default\_collator = (

DataCollatorWithPadding(processing\_class)

if processing\_class is not None

and isinstance(processing\_class, (PreTrainedTokenizerBase, SequenceFeatureExtractor))

else default\_data\_collator

)

self.data\_collator = data\_collator if data\_collator is not None else default\_collator

self.train\_dataset = train\_dataset

self.eval\_dataset = eval\_dataset

self.processing\_class = processing\_class

# Bnb Quantized models doesn't support `.to` operation.

if (

self.place\_model\_on\_device

and not getattr(model, "quantization\_method", None) == QuantizationMethod.BITS\_AND\_BYTES

):

self.\_move\_model\_to\_device(model, args.device)

# Force n\_gpu to 1 to avoid DataParallel as MP will manage the GPUs

if self.is\_model\_parallel:

self.args.\_n\_gpu = 1

# later use `self.model is self.model\_wrapped` to check if it's wrapped or not

self.model\_wrapped = model

self.model = model

# Just in case the model was wrapped outside of the `Trainer`

unwrapped\_model = self.accelerator.unwrap\_model(model)

model\_forward = (

unwrapped\_model.forward

if not \_is\_peft\_model(unwrapped\_model)

else unwrapped\_model.get\_base\_model().forward

)

forward\_params = inspect.signature(model\_forward).parameters

# Check if the model has explicit setup for loss kwargs,

# if not, check if `\*\*kwargs` are in model.forward

if hasattr(model, "accepts\_loss\_kwargs"):

self.model\_accepts\_loss\_kwargs = model.accepts\_loss\_kwargs

else:

self.model\_accepts\_loss\_kwargs = any(

k.kind == inspect.Parameter.VAR\_KEYWORD for k in forward\_params.values()

)

self.neftune\_noise\_alpha = args.neftune\_noise\_alpha

self.compute\_metrics = compute\_metrics

self.preprocess\_logits\_for\_metrics = preprocess\_logits\_for\_metrics

self.optimizer, self.lr\_scheduler = optimizers

self.optimizer\_cls\_and\_kwargs = optimizer\_cls\_and\_kwargs

if self.optimizer\_cls\_and\_kwargs is not None and self.optimizer is not None:

raise RuntimeError("Passing both `optimizers` and `optimizer\_cls\_and\_kwargs` arguments is incompatible.")

if model\_init is not None and (self.optimizer is not None or self.lr\_scheduler is not None):

raise RuntimeError(

"Passing a `model\_init` is incompatible with providing the `optimizers` argument. "

"You should subclass `Trainer` and override the `create\_optimizer\_and\_scheduler` method."

)

if is\_torch\_xla\_available() and self.optimizer is not None:

for param in self.model.parameters():

model\_device = param.device

break

for param\_group in self.optimizer.param\_groups:

if len(param\_group["params"]) > 0:

optimizer\_device = param\_group["params"][0].device

break

if model\_device != optimizer\_device:

raise ValueError(

"The model and the optimizer parameters are not on the same device, which probably means you"

" created an optimizer around your model \*\*before\*\* putting on the device and passing it to the"

" `Trainer`. Make sure the lines `import torch\_xla.core.xla\_model as xm` and"

" `model.to(xm.xla\_device())` is performed before the optimizer creation in your script."

)

if (self.is\_fsdp\_xla\_enabled or self.is\_fsdp\_enabled) and (

self.optimizer is not None or self.lr\_scheduler is not None

):

raise RuntimeError(

"Passing `optimizers` is not allowed if PyTorch FSDP is enabled. "

"You should subclass `Trainer` and override the `create\_optimizer\_and\_scheduler` method."

)

default\_callbacks = DEFAULT\_CALLBACKS + get\_reporting\_integration\_callbacks(self.args.report\_to)

callbacks = default\_callbacks if callbacks is None else default\_callbacks + callbacks

self.callback\_handler = CallbackHandler(

callbacks, self.model, self.processing\_class, self.optimizer, self.lr\_scheduler

)

self.add\_callback(PrinterCallback if self.args.disable\_tqdm else DEFAULT\_PROGRESS\_CALLBACK)

# Will be set to True by `self.\_setup\_loggers()` on first call to `self.log()`.

self.\_loggers\_initialized = False

# Create distant repo and output directory if needed

self.hub\_model\_id = None

if self.args.push\_to\_hub:

self.init\_hf\_repo()

if self.args.should\_save:

os.makedirs(self.args.output\_dir, exist\_ok=True)

if not callable(self.data\_collator) and callable(getattr(self.data\_collator, "collate\_batch", None)):

raise ValueError("The `data\_collator` should be a simple callable (function, class with `\_\_call\_\_`).")

if args.max\_steps > 0 and args.num\_train\_epochs > 0:

logger.info("max\_steps is given, it will override any value given in num\_train\_epochs")

if train\_dataset is not None and not has\_length(train\_dataset) and args.max\_steps <= 0:

raise ValueError(

"The train\_dataset does not implement \_\_len\_\_, max\_steps has to be specified. "

"The number of steps needs to be known in advance for the learning rate scheduler."

)

if (

train\_dataset is not None

and isinstance(train\_dataset, torch.utils.data.IterableDataset)

and args.group\_by\_length

):

raise ValueError("the `--group\_by\_length` option is only available for `Dataset`, not `IterableDataset")

self.\_signature\_columns = None

# Mixed precision setup

self.use\_apex = False

self.use\_cpu\_amp = False

# Mixed precision setup for SageMaker Model Parallel

if is\_sagemaker\_mp\_enabled():

# BF16 + model parallelism in SageMaker: currently not supported, raise an error

if args.bf16:

raise ValueError("SageMaker Model Parallelism does not support BF16 yet. Please use FP16 instead ")

if IS\_SAGEMAKER\_MP\_POST\_1\_10:

# When there's mismatch between SMP config and trainer argument, use SMP config as truth

if args.fp16 != smp.state.cfg.fp16:

logger.warning(

f"FP16 provided in SM\_HP\_MP\_PARAMETERS is {smp.state.cfg.fp16}, "

f"but FP16 provided in trainer argument is {args.fp16}, "

f"setting to {smp.state.cfg.fp16}"

)

args.fp16 = smp.state.cfg.fp16

else:

# smp < 1.10 does not support fp16 in trainer.

if hasattr(smp.state.cfg, "fp16"):

logger.warning(

f"FP16 provided in SM\_HP\_MP\_PARAMETERS is {smp.state.cfg.fp16}, "

"but SageMaker Model Parallelism < 1.10 does not support FP16 in trainer."

)

if (args.fp16 or args.bf16) and args.half\_precision\_backend == "auto":

if args.device == torch.device("cpu"):

if args.fp16:

if not is\_torch\_greater\_or\_equal\_than\_2\_3:

raise ValueError("Tried to use `fp16` but it is not supported on cpu")

else:

args.half\_precision\_backend = "cpu\_amp"

logger.info(f"Using {args.half\_precision\_backend} half precision backend")

if (args.fp16 or args.bf16) and not (self.is\_deepspeed\_enabled or is\_sagemaker\_mp\_enabled()):

# deepspeed and SageMaker Model Parallel manage their own half precision

if args.half\_precision\_backend == "cpu\_amp":

self.use\_cpu\_amp = True

self.amp\_dtype = torch.bfloat16

elif args.half\_precision\_backend == "apex":

if not is\_apex\_available():

raise ImportError(

"Using FP16 with APEX but APEX is not installed, please refer to"

" https://www.github.com/nvidia/apex."

)

self.use\_apex = True

# Label smoothing

if self.args.label\_smoothing\_factor != 0:

self.label\_smoother = LabelSmoother(epsilon=self.args.label\_smoothing\_factor)

else:

self.label\_smoother = None

self.control = TrainerControl()

self.state = TrainerState(

is\_local\_process\_zero=self.is\_local\_process\_zero(),

is\_world\_process\_zero=self.is\_world\_process\_zero(),

stateful\_callbacks=[

cb for cb in self.callback\_handler.callbacks + [self.control] if isinstance(cb, ExportableState)

],

)

# Internal variable to count flos in each process, will be accumulated in `self.state.total\_flos` then

# returned to 0 every time flos need to be logged

self.current\_flos = 0

self.hp\_search\_backend = None

if \_is\_peft\_model(self.model) and self.args.label\_names is None:

logger.warning(

f"No label\_names provided for model class `{self.model.\_\_class\_\_.\_\_name\_\_}`."

" Since `PeftModel` hides base models input arguments, if label\_names is not given, label\_names can't be set automatically within `Trainer`."

" Note that empty label\_names list will be used instead."

)

default\_label\_names = find\_labels(self.model.\_\_class\_\_)

self.label\_names = default\_label\_names if self.args.label\_names is None else self.args.label\_names

self.can\_return\_loss = can\_return\_loss(self.model.\_\_class\_\_)

self.control = self.callback\_handler.on\_init\_end(self.args, self.state, self.control)

# Internal variables to help with automatic batch size reduction

self.\_train\_batch\_size = args.train\_batch\_size

self.\_created\_lr\_scheduler = False

# very last

self.\_memory\_tracker.stop\_and\_update\_metrics()

self.is\_fsdp\_xla\_v2\_enabled = args.fsdp\_config.get("xla\_fsdp\_v2", False)

if self.is\_fsdp\_xla\_v2\_enabled:

if not IS\_XLA\_FSDPV2\_POST\_2\_2:

raise ValueError("FSDPv2 requires `torch\_xla` 2.2 or higher.")

# Prepare the SPMD mesh that is going to be used by the data loader and the FSDPv2 wrapper.

# Tensor axis is just a placeholder where it will not be used in FSDPv2.

num\_devices = xr.global\_runtime\_device\_count()

xs.set\_global\_mesh(xs.Mesh(np.array(range(num\_devices)), (num\_devices, 1), axis\_names=("fsdp", "tensor")))

self.is\_fsdp\_xla\_v1\_enabled = self.is\_fsdp\_xla\_enabled and not self.is\_fsdp\_xla\_v2\_enabled

@property

def tokenizer(self) -> Optional[PreTrainedTokenizerBase]:

logger.warning("Trainer.tokenizer is now deprecated. You should use Trainer.processing\_class instead.")

return self.processing\_class

@tokenizer.setter

def tokenizer(self, processing\_class) -> None:

logger.warning(

"Trainer.tokenizer is now deprecated. You should use `Trainer.processing\_class = processing\_class` instead."

)

self.processing\_class = processing\_class

def \_activate\_neftune(self, model):

r"""

Activates the neftune as presented in this code: https://github.com/neelsjain/NEFTune and paper:

https://arxiv.org/abs/2310.05914

"""

unwrapped\_model = self.accelerator.unwrap\_model(model)

if \_is\_peft\_model(unwrapped\_model):

embeddings = unwrapped\_model.base\_model.model.get\_input\_embeddings()

else:

embeddings = unwrapped\_model.get\_input\_embeddings()

del unwrapped\_model

embeddings.neftune\_noise\_alpha = self.neftune\_noise\_alpha

hook\_handle = embeddings.register\_forward\_hook(neftune\_post\_forward\_hook)

self.neftune\_hook\_handle = hook\_handle

return model

def \_deactivate\_neftune(self, model):

"""

Deactivates the neftune method. Make sure to call `\_activate\_neftune` first.

"""

if not hasattr(self, "neftune\_hook\_handle"):

raise ValueError("Neftune is not activated make sure to call `trainer.\_activate\_neftune()` first")

unwrapped\_model = self.accelerator.unwrap\_model(model)

if \_is\_peft\_model(unwrapped\_model):

embeddings = unwrapped\_model.base\_model.model.get\_input\_embeddings()

else:

embeddings = unwrapped\_model.get\_input\_embeddings()

self.neftune\_hook\_handle.remove()

del embeddings.neftune\_noise\_alpha, unwrapped\_model

def add\_callback(self, callback):

"""

Add a callback to the current list of [`~transformers.TrainerCallback`].

Args:

callback (`type` or [`~transformers.TrainerCallback]`):

A [`~transformers.TrainerCallback`] class or an instance of a [`~transformers.TrainerCallback`]. In the

first case, will instantiate a member of that class.

"""

self.callback\_handler.add\_callback(callback)

def pop\_callback(self, callback):

"""

Remove a callback from the current list of [`~transformers.TrainerCallback`] and returns it.

If the callback is not found, returns `None` (and no error is raised).

Args:

callback (`type` or [`~transformers.TrainerCallback]`):

A [`~transformers.TrainerCallback`] class or an instance of a [`~transformers.TrainerCallback`]. In the

first case, will pop the first member of that class found in the list of callbacks.

Returns:

[`~transformers.TrainerCallback`]: The callback removed, if found.

"""

return self.callback\_handler.pop\_callback(callback)

def remove\_callback(self, callback):

"""

Remove a callback from the current list of [`~transformers.TrainerCallback`].

Args:

callback (`type` or [`~transformers.TrainerCallback]`):

A [`~transformers.TrainerCallback`] class or an instance of a [`~transformers.TrainerCallback`]. In the

first case, will remove the first member of that class found in the list of callbacks.

"""

self.callback\_handler.remove\_callback(callback)

def \_move\_model\_to\_device(self, model, device):

model = model.to(device)

# Moving a model to an XLA device disconnects the tied weights, so we have to retie them.

if self.args.parallel\_mode == ParallelMode.TPU and hasattr(model, "tie\_weights"):

model.tie\_weights()

def \_set\_signature\_columns\_if\_needed(self):

if self.\_signature\_columns is None:

# Inspect model forward signature to keep only the arguments it accepts.

model\_to\_inspect = self.model

if \_is\_peft\_model(self.model):

if hasattr(self.model, "get\_base\_model"):

model\_to\_inspect = self.model.get\_base\_model()

else:

# PeftMixedModel do not provide a `get\_base\_model` method

model\_to\_inspect = self.model.base\_model.model

signature = inspect.signature(model\_to\_inspect.forward)

self.\_signature\_columns = list(signature.parameters.keys())

# Labels may be named label or label\_ids, the default data collator handles that.

self.\_signature\_columns += list(set(["label", "label\_ids"] + self.label\_names))

def \_remove\_unused\_columns(self, dataset: "datasets.Dataset", description: Optional[str] = None):

if not self.args.remove\_unused\_columns:

return dataset

self.\_set\_signature\_columns\_if\_needed()

signature\_columns = self.\_signature\_columns

ignored\_columns = list(set(dataset.column\_names) - set(signature\_columns))

if len(ignored\_columns) > 0:

dset\_description = "" if description is None else f"in the {description} set"

logger.info(

f"The following columns {dset\_description} don't have a corresponding argument in "

f"`{self.model.\_\_class\_\_.\_\_name\_\_}.forward` and have been ignored: {', '.join(ignored\_columns)}."

f" If {', '.join(ignored\_columns)} are not expected by `{self.model.\_\_class\_\_.\_\_name\_\_}.forward`, "

" you can safely ignore this message."

)

columns = [k for k in signature\_columns if k in dataset.column\_names]

if len(columns) == 0:

raise ValueError(

"No columns in the dataset match the model's forward method signature: ({', '.join(signature\_columns)}). "

f"The following columns have been ignored: [{', '.join(ignored\_columns)}]. "

"Please check the dataset and model. You may need to set `remove\_unused\_columns=False` in `TrainingArguments`."

)

if version.parse(datasets.\_\_version\_\_) < version.parse("1.4.0"):

dataset.set\_format(

type=dataset.format["type"], columns=columns, format\_kwargs=dataset.format["format\_kwargs"]

)

return dataset

else:

return dataset.remove\_columns(ignored\_columns)

def \_get\_collator\_with\_removed\_columns(

self, data\_collator: Callable, description: Optional[str] = None

) -> Callable:

"""Wrap the data collator in a callable removing unused columns."""

if not self.args.remove\_unused\_columns:

return data\_collator

self.\_set\_signature\_columns\_if\_needed()

signature\_columns = self.\_signature\_columns

remove\_columns\_collator = RemoveColumnsCollator(

data\_collator=data\_collator,

signature\_columns=signature\_columns,

logger=logger,

description=description,

model\_name=self.model.\_\_class\_\_.\_\_name\_\_,

)

return remove\_columns\_collator

def \_get\_train\_sampler(self) -> Optional[torch.utils.data.Sampler]:

if self.train\_dataset is None or not has\_length(self.train\_dataset):

return None

# Build the sampler.

if self.args.group\_by\_length:

if is\_datasets\_available() and isinstance(self.train\_dataset, datasets.Dataset):

lengths = (

self.train\_dataset[self.args.length\_column\_name]

if self.args.length\_column\_name in self.train\_dataset.column\_names

else None

)

else:

lengths = None

model\_input\_name = (

self.processing\_class.model\_input\_names[0] if self.processing\_class is not None else None

)

return LengthGroupedSampler(

self.args.train\_batch\_size \* self.args.gradient\_accumulation\_steps,

dataset=self.train\_dataset,

lengths=lengths,

model\_input\_name=model\_input\_name,

)

else:

return RandomSampler(self.train\_dataset)

def get\_train\_dataloader(self) -> DataLoader:

"""

Returns the training [`~torch.utils.data.DataLoader`].

Will use no sampler if `train\_dataset` does not implement `\_\_len\_\_`, a random sampler (adapted to distributed

training if necessary) otherwise.

Subclass and override this method if you want to inject some custom behavior.

"""

if self.train\_dataset is None:

raise ValueError("Trainer: training requires a train\_dataset.")

train\_dataset = self.train\_dataset

data\_collator = self.data\_collator

if is\_datasets\_available() and isinstance(train\_dataset, datasets.Dataset):

train\_dataset = self.\_remove\_unused\_columns(train\_dataset, description="training")

else:

data\_collator = self.\_get\_collator\_with\_removed\_columns(data\_collator, description="training")

dataloader\_params = {

"batch\_size": self.\_train\_batch\_size,

"collate\_fn": data\_collator,

"num\_workers": self.args.dataloader\_num\_workers,

"pin\_memory": self.args.dataloader\_pin\_memory,

"persistent\_workers": self.args.dataloader\_persistent\_workers,

}

if not isinstance(train\_dataset, torch.utils.data.IterableDataset):

dataloader\_params["sampler"] = self.\_get\_train\_sampler()

dataloader\_params["drop\_last"] = self.args.dataloader\_drop\_last

dataloader\_params["worker\_init\_fn"] = seed\_worker

dataloader\_params["prefetch\_factor"] = self.args.dataloader\_prefetch\_factor

return self.accelerator.prepare(DataLoader(train\_dataset, \*\*dataloader\_params))

def \_get\_eval\_sampler(self, eval\_dataset: Dataset) -> Optional[torch.utils.data.Sampler]:

if eval\_dataset is None or not has\_length(eval\_dataset):

return None

# Build the sampler.

# Deprecated code

if self.args.use\_legacy\_prediction\_loop:

if is\_torch\_xla\_available():

return SequentialDistributedSampler(

eval\_dataset, num\_replicas=xm.xrt\_world\_size(), rank=xm.get\_ordinal()

)

elif is\_sagemaker\_mp\_enabled():

return SequentialDistributedSampler(

eval\_dataset,

num\_replicas=smp.dp\_size(),

rank=smp.dp\_rank(),

batch\_size=self.args.per\_device\_eval\_batch\_size,

)

else:

return SequentialSampler(eval\_dataset)

if self.args.group\_by\_length:

if is\_datasets\_available() and isinstance(eval\_dataset, datasets.Dataset):

lengths = (

eval\_dataset[self.args.length\_column\_name]

if self.args.length\_column\_name in eval\_dataset.column\_names

else None

)

else:

lengths = None

model\_input\_name = (

self.processing\_class.model\_input\_names[0] if self.processing\_class is not None else None

)

return LengthGroupedSampler(

self.args.eval\_batch\_size,

dataset=eval\_dataset,

lengths=lengths,

model\_input\_name=model\_input\_name,

)

if self.args.world\_size <= 1:

return SequentialSampler(eval\_dataset)

else:

return None

def get\_eval\_dataloader(self, eval\_dataset: Optional[Union[str, Dataset]] = None) -> DataLoader:

"""

Returns the evaluation [`~torch.utils.data.DataLoader`].

Subclass and override this method if you want to inject some custom behavior.

Args:

eval\_dataset (`str` or `torch.utils.data.Dataset`, \*optional\*):

If a `str`, will use `self.eval\_dataset[eval\_dataset]` as the evaluation dataset. If a `Dataset`, will override `self.eval\_dataset` and must implement `\_\_len\_\_`. If it is a [`~datasets.Dataset`], columns not accepted by the `model.forward()` method are automatically removed.

"""

if eval\_dataset is None and self.eval\_dataset is None:

raise ValueError("Trainer: evaluation requires an eval\_dataset.")

# If we have persistent workers, don't do a fork bomb especially as eval datasets

# don't change during training

dataloader\_key = eval\_dataset if isinstance(eval\_dataset, str) else "eval"

if (

hasattr(self, "\_eval\_dataloaders")

and dataloader\_key in self.\_eval\_dataloaders

and self.args.dataloader\_persistent\_workers

):

return self.accelerator.prepare(self.\_eval\_dataloaders[dataloader\_key])

eval\_dataset = (

self.eval\_dataset[eval\_dataset]

if isinstance(eval\_dataset, str)

else eval\_dataset

if eval\_dataset is not None

else self.eval\_dataset

)

data\_collator = self.data\_collator

if is\_datasets\_available() and isinstance(eval\_dataset, datasets.Dataset):

eval\_dataset = self.\_remove\_unused\_columns(eval\_dataset, description="evaluation")

else:

data\_collator = self.\_get\_collator\_with\_removed\_columns(data\_collator, description="evaluation")

dataloader\_params = {

"batch\_size": self.args.eval\_batch\_size,

"collate\_fn": data\_collator,

"num\_workers": self.args.dataloader\_num\_workers,

"pin\_memory": self.args.dataloader\_pin\_memory,

"persistent\_workers": self.args.dataloader\_persistent\_workers,

}

if not isinstance(eval\_dataset, torch.utils.data.IterableDataset):

dataloader\_params["sampler"] = self.\_get\_eval\_sampler(eval\_dataset)

dataloader\_params["drop\_last"] = self.args.dataloader\_drop\_last

dataloader\_params["prefetch\_factor"] = self.args.dataloader\_prefetch\_factor

# accelerator.free\_memory() will destroy the references, so

# we need to store the non-prepared version

eval\_dataloader = DataLoader(eval\_dataset, \*\*dataloader\_params)

if self.args.dataloader\_persistent\_workers:

if hasattr(self, "\_eval\_dataloaders"):

self.\_eval\_dataloaders[dataloader\_key] = eval\_dataloader

else:

self.\_eval\_dataloaders = {dataloader\_key: eval\_dataloader}

return self.accelerator.prepare(eval\_dataloader)

def get\_test\_dataloader(self, test\_dataset: Dataset) -> DataLoader:

"""

Returns the test [`~torch.utils.data.DataLoader`].

Subclass and override this method if you want to inject some custom behavior.

Args:

test\_dataset (`torch.utils.data.Dataset`, \*optional\*):

The test dataset to use. If it is a [`~datasets.Dataset`], columns not accepted by the

`model.forward()` method are automatically removed. It must implement `\_\_len\_\_`.

"""

data\_collator = self.data\_collator

if is\_datasets\_available() and isinstance(test\_dataset, datasets.Dataset):

test\_dataset = self.\_remove\_unused\_columns(test\_dataset, description="test")

else:

data\_collator = self.\_get\_collator\_with\_removed\_columns(data\_collator, description="test")

dataloader\_params = {

"batch\_size": self.args.eval\_batch\_size,

"collate\_fn": data\_collator,

"num\_workers": self.args.dataloader\_num\_workers,

"pin\_memory": self.args.dataloader\_pin\_memory,

"persistent\_workers": self.args.dataloader\_persistent\_workers,

}

if not isinstance(test\_dataset, torch.utils.data.IterableDataset):

dataloader\_params["sampler"] = self.\_get\_eval\_sampler(test\_dataset)

dataloader\_params["drop\_last"] = self.args.dataloader\_drop\_last

dataloader\_params["prefetch\_factor"] = self.args.dataloader\_prefetch\_factor

# We use the same batch\_size as for eval.

return self.accelerator.prepare(DataLoader(test\_dataset, \*\*dataloader\_params))

def create\_optimizer\_and\_scheduler(self, num\_training\_steps: int):

"""

Setup the optimizer and the learning rate scheduler.

We provide a reasonable default that works well. If you want to use something else, you can pass a tuple in the

Trainer's init through `optimizers`, or subclass and override this method (or `create\_optimizer` and/or

`create\_scheduler`) in a subclass.

"""

self.create\_optimizer()

if IS\_SAGEMAKER\_MP\_POST\_1\_10 and smp.state.cfg.fp16:

# If smp >= 1.10 and fp16 is enabled, we unwrap the optimizer

optimizer = self.optimizer.optimizer

else:

optimizer = self.optimizer

self.create\_scheduler(num\_training\_steps=num\_training\_steps, optimizer=optimizer)

def get\_decay\_parameter\_names(self, model) -> list[str]:

"""

Get all parameter names that weight decay will be applied to.

This function filters out parameters in two ways:

1. By layer type (instances of layers specified in ALL\_LAYERNORM\_LAYERS)

2. By parameter name patterns (containing 'bias', 'layernorm', or 'rmsnorm')

"""

decay\_parameters = get\_parameter\_names(model, ALL\_LAYERNORM\_LAYERS, ["bias", "layernorm", "rmsnorm"])

return decay\_parameters

def create\_optimizer(self):

"""

Setup the optimizer.

We provide a reasonable default that works well. If you want to use something else, you can pass a tuple in the

Trainer's init through `optimizers`, or subclass and override this method in a subclass.

"""

opt\_model = self.model\_wrapped if is\_sagemaker\_mp\_enabled() else self.model

if self.optimizer is None:

decay\_parameters = self.get\_decay\_parameter\_names(opt\_model)

optimizer\_grouped\_parameters = [

{

"params": [

p for n, p in opt\_model.named\_parameters() if (n in decay\_parameters and p.requires\_grad)

],

"weight\_decay": self.args.weight\_decay,

},

{

"params": [

p for n, p in opt\_model.named\_parameters() if (n not in decay\_parameters and p.requires\_grad)

],

"weight\_decay": 0.0,

},

]

if self.optimizer\_cls\_and\_kwargs is not None:

optimizer\_cls, optimizer\_kwargs = self.optimizer\_cls\_and\_kwargs

else:

optimizer\_cls, optimizer\_kwargs = self.get\_optimizer\_cls\_and\_kwargs(self.args, opt\_model)

# Overwrite `params` in case it's created by `get\_optimizer\_cls\_and\_kwargs`

# e.g. for GaLore optimizer.

if "params" in optimizer\_kwargs:

optimizer\_grouped\_parameters = optimizer\_kwargs.pop("params")

# Overwrite `model` in case it's created by `get\_optimizer\_cls\_and\_kwargs`

# e.g. for LOMO optimizer.

if "model" in optimizer\_kwargs:

optimizer\_grouped\_parameters = optimizer\_kwargs.pop("model")

# For layer-wise dummy optimizers we overwrite optimizer\_grouped\_parameters with `optimizer\_dict`

# to avoid arguments conflicts.

if "optimizer\_dict" in optimizer\_kwargs:

optimizer\_grouped\_parameters = optimizer\_kwargs.pop("optimizer\_dict")

self.optimizer = optimizer\_cls(optimizer\_grouped\_parameters, \*\*optimizer\_kwargs)

if optimizer\_cls.\_\_name\_\_ == "Adam8bit":

import bitsandbytes

manager = bitsandbytes.optim.GlobalOptimManager.get\_instance()

skipped = 0

for module in opt\_model.modules():

if isinstance(module, nn.Embedding):

skipped += sum({p.data\_ptr(): p.numel() for p in module.parameters()}.values())

logger.info(f"skipped {module}: {skipped / 2\*\*20}M params")

manager.register\_module\_override(module, "weight", {"optim\_bits": 32})

logger.debug(f"bitsandbytes: will optimize {module} in fp32")

logger.info(f"skipped: {skipped / 2\*\*20}M params")

if is\_sagemaker\_mp\_enabled():

self.optimizer = smp.DistributedOptimizer(self.optimizer)

return self.optimizer

def get\_num\_trainable\_parameters(self):

"""

Get the number of trainable parameters.

"""

return sum(p.numel() for p in self.model.parameters() if p.requires\_grad)

def get\_learning\_rates(self):

"""

Returns the learning rate of each parameter from self.optimizer.

"""

if self.optimizer is None:

raise ValueError("Trainer optimizer is None, please make sure you have setup the optimizer before.")

return [group["lr"] for group in self.optimizer.param\_groups]

def get\_optimizer\_group(self, param: Optional[Union[str, torch.nn.parameter.Parameter]] = None):

"""

Returns optimizer group for a parameter if given, else returns all optimizer groups for params.

Args:

param (`str` or `torch.nn.parameter.Parameter`, \*optional\*):

The parameter for which optimizer group needs to be returned.

"""

if self.optimizer is None:

raise ValueError("Trainer optimizer is None, please make sure you have setup the optimizer before.")

if param is not None:

for group in self.optimizer.param\_groups:

if param in group["params"]:

return group

return [group["params"] for group in self.optimizer.param\_groups]

@staticmethod

def get\_optimizer\_cls\_and\_kwargs(

args: TrainingArguments, model: Optional[PreTrainedModel] = None

) -> tuple[Any, Any]:

"""

Returns the optimizer class and optimizer parameters based on the training arguments.

Args:

args (`transformers.training\_args.TrainingArguments`):

The training arguments for the training session.

"""

# parse args.optim\_args

optim\_args = {}

if args.optim\_args:

for mapping in args.optim\_args.replace(" ", "").split(","):

key, value = mapping.split("=")

optim\_args[key] = value

optimizer\_kwargs = {"lr": args.learning\_rate}

adam\_kwargs = {

"betas": (args.adam\_beta1, args.adam\_beta2),

"eps": args.adam\_epsilon,

}

def setup\_low\_rank\_optimizer(

optimizer\_name: str,

optimizer\_mapping: dict[str, Any],

optim\_kwargs: dict[str, Any],

is\_layerwise\_supported: bool = True,

) -> tuple[Any, Any]:

"""

Helper function to set up low-rank optimizers like GaLore and Apollo.

Args:

optimizer\_name (str): Name of the optimizer.

optimizer\_mapping (dict): Mapping of optimizer names to their classes.

optim\_kwargs (dict): Keyword arguments for the optimizer.

is\_layerwise\_supported (bool): Whether layerwise optimization is supported.

Returns:

Tuple[Any, Any]: Optimizer class and updated optimizer kwargs.

"""

is\_layerwise = optimizer\_name.lower().endswith("layerwise")

if is\_layerwise and args.parallel\_mode == ParallelMode.DISTRIBUTED and is\_layerwise\_supported:

raise NotImplementedError(f"Layer-wise {optimizer\_name} does not support DDP at this time")

optimizer\_cls = optimizer\_mapping[optimizer\_name]

if args.optim\_target\_modules is None:

raise ValueError(f"You need to define `optim\_target\_modules` to use {optimizer\_name} optimizers")

if not isinstance(args.optim\_target\_modules, (list, str)):

raise ValueError(

f"`optim\_target\_modules` must be a list of strings, a regex string, or 'all-linear'. Got: {args.optim\_target\_modules}"

)

if model is None:

raise ValueError(f"You need to pass a model to initialize {optimizer\_name} optimizer.")

all\_linear = (

isinstance(args.optim\_target\_modules, str)

and args.optim\_target\_modules.replace("\_", "-") == "all-linear"

)

target\_params\_names = []

for module\_name, module in model.named\_modules():

target\_module\_exists, is\_regex = check\_target\_module\_exists(

args.optim\_target\_modules, module\_name, return\_is\_regex=True

)

if not isinstance(module, nn.Linear):

if target\_module\_exists and not is\_regex:

logger.warning(

f"{module\_name} matched but ignored. {optimizer\_name} only supports linear layers."

)

continue

if not target\_module\_exists and not all\_linear:

continue

target\_params\_names.append(module\_name + ".weight")

if len(target\_params\_names) == 0:

raise ValueError(f"No target modules found for {optimizer\_name} ({args.optim\_target\_modules}).")

target\_params = [p for n, p in model.named\_parameters() if n in target\_params\_names]

non\_target\_params = [p for n, p in model.named\_parameters() if n not in target\_params\_names]

optim\_kwargs.update(optim\_args)

param\_groups = [

{"params": non\_target\_params},

{"params": target\_params, \*\*optim\_kwargs},

]

if is\_layerwise:

if args.gradient\_accumulation\_steps != 1:

raise ValueError(f"Layerwise {optimizer\_name} does not support gradient accumulation!")

optimizer\_dict = {}

for param in non\_target\_params:

optimizer\_dict[param] = optimizer\_cls([{"params": [param]}], \*\*optimizer\_kwargs)

for param in target\_params:

optimizer\_dict[param] = optimizer\_cls([{"params": [param], \*\*optim\_kwargs}], \*\*optimizer\_kwargs)

def optimizer\_hook(param):

if param.grad is not None:

optimizer\_dict[param].step()

optimizer\_dict[param].zero\_grad()

for param in model.parameters():

if param.requires\_grad:

param.register\_post\_accumulate\_grad\_hook(optimizer\_hook)

optimizer\_cls = LayerWiseDummyOptimizer

optimizer\_kwargs.update({"optimizer\_dict": optimizer\_dict})

optimizer\_kwargs.update({"params": param\_groups})

return optimizer\_cls, optimizer\_kwargs

if args.optim == OptimizerNames.ADAFACTOR:

optimizer\_cls = Adafactor

optimizer\_kwargs.update({"scale\_parameter": False, "relative\_step": False})

elif args.optim in [OptimizerNames.ADAMW\_TORCH, OptimizerNames.ADAMW\_TORCH\_FUSED]:

from torch.optim import AdamW

optimizer\_cls = AdamW

optimizer\_kwargs.update(adam\_kwargs)

if args.optim == OptimizerNames.ADAMW\_TORCH\_FUSED:

optimizer\_kwargs.update({"fused": True})

elif args.optim == OptimizerNames.ADAMW\_TORCH\_XLA:

try:

from torch\_xla.amp.syncfree import AdamW

optimizer\_cls = AdamW

optimizer\_kwargs.update(adam\_kwargs)

except ImportError:

raise ValueError("Trainer failed to import syncfree AdamW from torch\_xla.")

elif args.optim == OptimizerNames.ADAMW\_TORCH\_NPU\_FUSED:

try:

from torch\_npu.optim import NpuFusedAdamW

optimizer\_cls = NpuFusedAdamW

optimizer\_kwargs.update(adam\_kwargs)

except ImportError:

raise ValueError("Trainer failed to import FusedAdamW from torch\_npu.")

elif args.optim == OptimizerNames.ADAMW\_APEX\_FUSED:

try:

from apex.optimizers import FusedAdam

optimizer\_cls = FusedAdam

optimizer\_kwargs.update(adam\_kwargs)

except ImportError:

raise ValueError("Trainer tried to instantiate apex FusedAdam but apex is not installed!")

elif args.optim in [

OptimizerNames.ADAMW\_BNB,

OptimizerNames.ADAMW\_8BIT,

OptimizerNames.PAGED\_ADAMW,

OptimizerNames.PAGED\_ADAMW\_8BIT,

OptimizerNames.ADEMAMIX,

OptimizerNames.ADEMAMIX\_8BIT,

OptimizerNames.PAGED\_ADEMAMIX,

OptimizerNames.PAGED\_ADEMAMIX\_8BIT,

OptimizerNames.LION,

OptimizerNames.LION\_8BIT,

OptimizerNames.PAGED\_LION,

OptimizerNames.PAGED\_LION\_8BIT,

OptimizerNames.RMSPROP\_BNB,

OptimizerNames.RMSPROP\_8BIT,

OptimizerNames.RMSPROP\_32BIT,

]:

try:

from bitsandbytes.optim import AdamW, Lion, RMSprop

is\_paged = False

optim\_bits = 32

optimizer\_cls = None

additional\_optim\_kwargs = adam\_kwargs

if "paged" in args.optim:

is\_paged = True

if "8bit" in args.optim:

optim\_bits = 8

if "adam" in args.optim:

optimizer\_cls = AdamW

elif "lion" in args.optim:

optimizer\_cls = Lion

additional\_optim\_kwargs = {"betas": (args.adam\_beta1, args.adam\_beta2)}

elif "rmsprop" in args.optim:

optimizer\_cls = RMSprop

# Above we pass all `adam\_kwargs` to the optimizer, here

# we only pass `optim\_args` which can be passed by the user.

additional\_optim\_kwargs = optim\_args

elif "ademamix" in args.optim:

if is\_bitsandbytes\_available() and version.parse(

importlib.metadata.version("bitsandbytes")

) < version.parse("0.44.0"):

raise ValueError(

"The AdEMAMix optimizer is not supported by your current version of `bitsandbytes`. "

"Please install `bitsandbytes` >= 0.44.0."

)

from bitsandbytes.optim import AdEMAMix

optimizer\_cls = AdEMAMix

additional\_optim\_kwargs = {

"betas": (

float(optim\_args.get("beta1", args.adam\_beta1)),

float(optim\_args.get("beta2", args.adam\_beta2)),

float(optim\_args.get("beta3", 0.9999)),

),

"alpha": float(optim\_args.get("alpha", 5.0)),

"eps": float(optim\_args.get("eps", args.adam\_epsilon)),

}

if "t\_alpha" in optim\_args:

additional\_optim\_kwargs["t\_alpha"] = int(optim\_args["t\_alpha"])

if "t\_beta3" in optim\_args:

additional\_optim\_kwargs["t\_beta3"] = int(optim\_args["t\_beta3"])

bnb\_kwargs = {"optim\_bits": optim\_bits}

if "rmsprop" not in args.optim:

bnb\_kwargs["is\_paged"] = is\_paged

optimizer\_kwargs.update(additional\_optim\_kwargs)

optimizer\_kwargs.update(bnb\_kwargs)

except ImportError:

raise ValueError("Trainer tried to instantiate bnb optimizer but `bitsandbytes` is not installed!")

if is\_bitsandbytes\_available() and version.parse(

importlib.metadata.version("bitsandbytes")

) < version.parse("0.41.1"):

logger.warning(

"You are using 8-bit optimizers with a version of `bitsandbytes` < 0.41.1. "

"It is recommended to update your version as a major bug has been fixed in 8-bit optimizers."

)

elif args.optim == OptimizerNames.ADAMW\_ANYPRECISION:

try:

from torchdistx.optimizers import AnyPrecisionAdamW

optimizer\_cls = AnyPrecisionAdamW

optimizer\_kwargs.update(adam\_kwargs)

# TODO Change dtypes back to M=FP32, Var = BF16, Kahan = False once they can be cast together in torchdistx.

optimizer\_kwargs.update(

{

"use\_kahan\_summation": strtobool(optim\_args.get("use\_kahan\_summation", "False")),

"momentum\_dtype": getattr(torch, optim\_args.get("momentum\_dtype", "float32")),

"variance\_dtype": getattr(torch, optim\_args.get("variance\_dtype", "float32")),

"compensation\_buffer\_dtype": getattr(

torch, optim\_args.get("compensation\_buffer\_dtype", "bfloat16")

),

}

)

except ImportError:

raise ValueError("Please install https://github.com/pytorch/torchdistx")

elif args.optim == OptimizerNames.SGD:

optimizer\_cls = torch.optim.SGD

elif args.optim == OptimizerNames.ADAGRAD:

optimizer\_cls = torch.optim.Adagrad

elif args.optim == OptimizerNames.RMSPROP:

optimizer\_cls = torch.optim.RMSprop

elif args.optim in [

OptimizerNames.GALORE\_ADAMW,

OptimizerNames.GALORE\_ADAMW\_8BIT,

OptimizerNames.GALORE\_ADAFACTOR,

OptimizerNames.GALORE\_ADAMW\_LAYERWISE,

OptimizerNames.GALORE\_ADAMW\_8BIT\_LAYERWISE,

OptimizerNames.GALORE\_ADAFACTOR\_LAYERWISE,

]:

if not is\_galore\_torch\_available():

raise ImportError(

"You need to install `galore\_torch` in order to use GaLore optimizers"

" install it with `pip install git+https://github.com/jiaweizzhao/GaLore`"

)

from galore\_torch import GaLoreAdafactor, GaLoreAdamW, GaLoreAdamW8bit

optimizer\_mapping = {

OptimizerNames.GALORE\_ADAMW: GaLoreAdamW,

OptimizerNames.GALORE\_ADAMW\_8BIT: GaLoreAdamW8bit,

OptimizerNames.GALORE\_ADAFACTOR: GaLoreAdafactor,

OptimizerNames.GALORE\_ADAMW\_LAYERWISE: GaLoreAdamW,

OptimizerNames.GALORE\_ADAMW\_8BIT\_LAYERWISE: GaLoreAdamW8bit,

OptimizerNames.GALORE\_ADAFACTOR\_LAYERWISE: GaLoreAdafactor,

}

galore\_optim\_kwargs = {

"rank": int(optim\_args.pop("rank", 128)),

"update\_proj\_gap": int(optim\_args.pop("update\_proj\_gap", 200)),

"scale": float(optim\_args.pop("scale", 0.25)),

"proj\_type": optim\_args.pop("proj\_type", "std"),

}

optimizer\_cls, optimizer\_kwargs = setup\_low\_rank\_optimizer(

args.optim, optimizer\_mapping, galore\_optim\_kwargs

)

if args.optim == OptimizerNames.GALORE\_ADAFACTOR:

optimizer\_kwargs.update({"scale\_parameter": False, "relative\_step": False})

elif args.optim in [

OptimizerNames.APOLLO\_ADAMW,

OptimizerNames.APOLLO\_ADAMW\_LAYERWISE,

]:

if not is\_apollo\_torch\_available():

raise ImportError(

"You need to install `apollo\_torch` in order to use APOLLO optimizers"

" install it with `pip install git+https://github.com/zhuhanqing/APOLLO`"

)

from apollo\_torch import APOLLOAdamW

optimizer\_mapping = {

OptimizerNames.APOLLO\_ADAMW: APOLLOAdamW,

OptimizerNames.APOLLO\_ADAMW\_LAYERWISE: APOLLOAdamW,

}

apollo\_optim\_kwargs = {

"rank": int(optim\_args.pop("rank", 128)),

"proj": optim\_args.pop("proj", "random"),

"scale\_type": optim\_args.pop("scale\_type", "channel"),

"update\_proj\_gap": int(optim\_args.pop("update\_proj\_gap", 200)),

"scale": float(optim\_args.pop("scale", 1.0)),

"proj\_type": optim\_args.pop("proj\_type", "std"),

}

optimizer\_cls, optimizer\_kwargs = setup\_low\_rank\_optimizer(

args.optim, optimizer\_mapping, apollo\_optim\_kwargs

)

elif args.optim in [OptimizerNames.LOMO, OptimizerNames.ADALOMO]:

if not is\_lomo\_available():

raise ImportError(

"You need to install `lomo\_optim` in order to use LOMO optimizers"

" install it with `pip install lomo-optim`"

)

if not is\_accelerate\_available("0.30.0"):

raise ImportError("You need to have `accelerate>=0.30.0` to be able to use LOMO optimizers")

if model is None:

raise ValueError("You need to pass a `model` in order to correctly initialize a LOMO optimizer.")

from lomo\_optim import AdaLomo, Lomo

if "ada" in args.optim:

optimizer\_cls = AdaLomo

else:

optimizer\_cls = Lomo

optimizer\_kwargs.update({"model": model})

elif args.optim == OptimizerNames.GROKADAMW:

if not is\_grokadamw\_available():

raise ValueError("Please install grokadamw with `pip install grokadamw`")

from grokadamw import GrokAdamW

optimizer\_cls = GrokAdamW

optimizer\_kwargs.update(

{

"alpha\_init": float(optim\_args.get("alpha\_init", 0.98)),

"lamb": float(optim\_args.get("lamb", 2.0)),

"gamma": float(optim\_args.get("gamma", 0.1)),

"grokking\_signal\_decay\_rate": float(optim\_args.get("grokking\_signal\_decay\_rate", 0.1)),

"gradient\_clipping": float(optim\_args.get("gradient\_clipping", 1.0)),

}

)

elif args.optim in [

OptimizerNames.ADAMW\_TORCH\_4BIT,

OptimizerNames.ADAMW\_TORCH\_8BIT,

]:

if not is\_torchao\_available() or version.parse(importlib.metadata.version("torchao")) < version.parse(

"0.4.0"

):

raise ImportError(

"You need to have `torchao>=0.4.0` in order to use torch 4-bit optimizers."

"Install it with `pip install torchao` or follow the instructions here: https://github.com/pytorch/ao"

)

if version.parse(importlib.metadata.version("torch")) <= version.parse("2.4"):

raise ImportError(

"You need to have `torch>2.4` in order to use torch 4-bit optimizers. "

"Install it with `pip install --upgrade torch` it is available on pipy. Otherwise, you need to install torch nightly."

)

from torchao.prototype.low\_bit\_optim import AdamW4bit, AdamW8bit

if args.optim == OptimizerNames.ADAMW\_TORCH\_4BIT:

optimizer\_cls = AdamW4bit

elif args.optim == OptimizerNames.ADAMW\_TORCH\_8BIT:

optimizer\_cls = AdamW8bit

else:

raise ValueError("Invalid optimizer")

optimizer\_kwargs.update(adam\_kwargs)

elif args.optim in [

OptimizerNames.SCHEDULE\_FREE\_RADAM,

OptimizerNames.SCHEDULE\_FREE\_ADAMW,

OptimizerNames.SCHEDULE\_FREE\_SGD,

]:

if not is\_schedulefree\_available():

raise ImportError(

"You need to install `schedulefree` in order to use schedulefree optimizers. "

"Install it with `pip install schedulefree.`"

)

if not is\_accelerate\_available("0.30.0"):

raise ImportError("You need to have `accelerate>=0.30.0` to be able to use schedulefree optimizers")

from schedulefree import AdamWScheduleFree, SGDScheduleFree

additional\_optim\_kwargs = {}

require\_warmup = True

if args.optim == OptimizerNames.SCHEDULE\_FREE\_RADAM:

if not is\_schedulefree\_available("1.4.0"):

raise ImportError(

"You need to install `schedulefree>=1.4.0` in order to use RAdamScheduleFree optimizer. "

"Install it with `pip install schedulefree.`"

)

from schedulefree import RAdamScheduleFree

optimizer\_cls = RAdamScheduleFree

additional\_optim\_kwargs = adam\_kwargs

require\_warmup = False

elif args.optim == OptimizerNames.SCHEDULE\_FREE\_ADAMW:

optimizer\_cls = AdamWScheduleFree

additional\_optim\_kwargs = adam\_kwargs

elif args.optim == OptimizerNames.SCHEDULE\_FREE\_SGD:

optimizer\_cls = SGDScheduleFree

else:

raise ValueError("Invalid schedulefree optimizer")

additional\_optim\_kwargs["weight\_decay"] = args.weight\_decay

if require\_warmup:

additional\_optim\_kwargs["warmup\_steps"] = args.warmup\_steps

additional\_optim\_kwargs.update(

{

"weight\_lr\_power": float(optim\_args.get("weight\_lr\_power", 2.0)),

"r": float(optim\_args.get("r", 0.0)),

}

)

optimizer\_kwargs.update(additional\_optim\_kwargs)

else:

raise ValueError(f"Trainer cannot instantiate unsupported optimizer: {args.optim}")

return optimizer\_cls, optimizer\_kwargs

def create\_scheduler(self, num\_training\_steps: int, optimizer: torch.optim.Optimizer = None):

"""

Setup the scheduler. The optimizer of the trainer must have been set up either before this method is called or

passed as an argument.

Args:

num\_training\_steps (int): The number of training steps to do.

"""

if self.lr\_scheduler is None:

self.lr\_scheduler = get\_scheduler(

self.args.lr\_scheduler\_type,

optimizer=self.optimizer if optimizer is None else optimizer,

num\_warmup\_steps=self.args.get\_warmup\_steps(num\_training\_steps),

num\_training\_steps=num\_training\_steps,

scheduler\_specific\_kwargs=self.args.lr\_scheduler\_kwargs,

)

self.\_created\_lr\_scheduler = True

return self.lr\_scheduler

def num\_examples(self, dataloader: DataLoader) -> int:

"""

Helper to get number of samples in a [`~torch.utils.data.DataLoader`] by accessing its dataset. When

dataloader.dataset does not exist or has no length, estimates as best it can

"""

try:

dataset = dataloader.dataset

# Special case for IterableDatasetShard, we need to dig deeper

if isinstance(dataset, IterableDatasetShard):

return len(dataloader.dataset.dataset)

return len(dataloader.dataset)

except (NameError, AttributeError, TypeError): # no dataset or length, estimate by length of dataloader

return len(dataloader) \* self.args.per\_device\_train\_batch\_size

@staticmethod

def num\_tokens(train\_dl: DataLoader, max\_steps: Optional[int] = None) -> int:

"""

Helper to get number of tokens in a [`~torch.utils.data.DataLoader`] by enumerating dataloader.

"""

train\_tokens = 0

try:

for batch in train\_dl:

tokens = batch["input\_ids"].numel()

if max\_steps is not None:

return tokens \* max\_steps

train\_tokens += tokens

except KeyError:

logger.warning("Cannot get num\_tokens from dataloader")

return train\_tokens

…

<More instance methods related to various HW components and Parallel architectures supported by the model>

### Trainer::train

def train(

self,

resume\_from\_checkpoint: Optional[Union[str, bool]] = None,

trial: Union["optuna.Trial", dict[str, Any], None] = None,

ignore\_keys\_for\_eval: Optional[list[str]] = None,

\*\*kwargs,

):

"""

Main training entry point.

Args:

resume\_from\_checkpoint (`str` or `bool`, \*optional\*):

If a `str`, local path to a saved checkpoint as saved by a previous instance of [`Trainer`]. If a

`bool` and equals `True`, load the last checkpoint in \*args.output\_dir\* as saved by a previous instance

of [`Trainer`]. If present, training will resume from the model/optimizer/scheduler states loaded here.

trial (`optuna.Trial` or `Dict[str, Any]`, \*optional\*):

The trial run or the hyperparameter dictionary for hyperparameter search.

ignore\_keys\_for\_eval (`List[str]`, \*optional\*)

A list of keys in the output of your model (if it is a dictionary) that should be ignored when

gathering predictions for evaluation during the training.

kwargs (`Dict[str, Any]`, \*optional\*):

Additional keyword arguments used to hide deprecated arguments

"""

if resume\_from\_checkpoint is False:

resume\_from\_checkpoint = None

# memory metrics - must set up as early as possible

self.\_memory\_tracker.start()

args = self.args

self.is\_in\_train = True

# Attach NEFTune hooks if necessary

if self.neftune\_noise\_alpha is not None:

self.model = self.\_activate\_neftune(self.model)

# do\_train is not a reliable argument, as it might not be set and .train() still called, so

# the following is a workaround:

if (

(args.fp16\_full\_eval or args.bf16\_full\_eval)

and not args.do\_train

and not self.is\_model\_parallel

and self.model\_init is None

):

self.\_move\_model\_to\_device(self.model, args.device)

if "model\_path" in kwargs:

resume\_from\_checkpoint = kwargs.pop("model\_path")

warnings.warn(

"`model\_path` is deprecated and will be removed in a future version. Use `resume\_from\_checkpoint` "

"instead.",

FutureWarning,

)

if len(kwargs) > 0:

raise TypeError(f"train() got unexpected keyword arguments: {', '.join(list(kwargs.keys()))}.")

# This might change the seed so needs to run first.

self.\_hp\_search\_setup(trial)

self.\_train\_batch\_size = self.args.train\_batch\_size

# Model re-init

model\_reloaded = False

if self.model\_init is not None:

# Seed must be set before instantiating the model when using model\_init.

enable\_full\_determinism(self.args.seed) if self.args.full\_determinism else set\_seed(self.args.seed)

self.model = self.call\_model\_init(trial)

model\_reloaded = True

# Reinitializes optimizer and scheduler

self.optimizer, self.lr\_scheduler = None, None

# Load potential model checkpoint

if isinstance(resume\_from\_checkpoint, bool) and resume\_from\_checkpoint:

resume\_from\_checkpoint = get\_last\_checkpoint(args.output\_dir)

if resume\_from\_checkpoint is None:

raise ValueError(f"No valid checkpoint found in output directory ({args.output\_dir})")

if resume\_from\_checkpoint is not None:

if not is\_sagemaker\_mp\_enabled() and not self.is\_deepspeed\_enabled and not self.is\_fsdp\_enabled:

self.\_load\_from\_checkpoint(resume\_from\_checkpoint)

# In case of repeating the find\_executable\_batch\_size, set `self.\_train\_batch\_size` properly

state = TrainerState.load\_from\_json(os.path.join(resume\_from\_checkpoint, TRAINER\_STATE\_NAME))

if state.train\_batch\_size is not None:

self.\_train\_batch\_size = state.train\_batch\_size

# If model was re-initialized, put it on the right device and update self.model\_wrapped

if model\_reloaded:

if self.place\_model\_on\_device:

self.\_move\_model\_to\_device(self.model, args.device)

self.model\_wrapped = self.model

inner\_training\_loop = find\_executable\_batch\_size(

self.\_inner\_training\_loop, self.\_train\_batch\_size, args.auto\_find\_batch\_size

)

if args.push\_to\_hub:

try:

# Disable progress bars when uploading models during checkpoints to avoid polluting stdout

hf\_hub\_utils.disable\_progress\_bars()

return inner\_training\_loop(

args=args,

resume\_from\_checkpoint=resume\_from\_checkpoint,

trial=trial,

ignore\_keys\_for\_eval=ignore\_keys\_for\_eval,

)

finally:

hf\_hub\_utils.enable\_progress\_bars()

else:

return inner\_training\_loop(

args=args,

resume\_from\_checkpoint=resume\_from\_checkpoint,

trial=trial,

ignore\_keys\_for\_eval=ignore\_keys\_for\_eval,

)

### find\_executable\_batch\_size in trainer\_utils.py

def find\_executable\_batch\_size(

function: callable = None, starting\_batch\_size: int = 128, auto\_find\_batch\_size: bool = False

):

"""

Args:

A basic decorator that will try to execute `function`. If it fails from exceptions related to out-of-memory or

CUDNN, the batch size is cut in half and passed to `function`. `function` must take in a `batch\_size` parameter as

its first argument.

function (`callable`, \*optional\*)

A function to wrap

starting\_batch\_size (`int`, \*optional\*)

The batch size to try and fit into memory

auto\_find\_batch\_size (`bool`, \*optional\*)

If False, will just execute `function`

"""

if function is None:

return functools.partial(

find\_executable\_batch\_size,

starting\_batch\_size=starting\_batch\_size,

auto\_find\_batch\_size=auto\_find\_batch\_size,

)

if auto\_find\_batch\_size:

requires\_backends(find\_executable\_batch\_size, "accelerate")

from accelerate.utils import find\_executable\_batch\_size as accelerate\_find\_executable\_batch\_size

return accelerate\_find\_executable\_batch\_size(function=function, starting\_batch\_size=starting\_batch\_size)

return functools.partial(function, batch\_size=starting\_batch\_size)

### Trainer::\_inner\_training\_loop

def \_inner\_training\_loop(

self, batch\_size=None, args=None, resume\_from\_checkpoint=None, trial=None, ignore\_keys\_for\_eval=None

):

self.accelerator.free\_memory()

self.\_train\_batch\_size = batch\_size

if self.args.auto\_find\_batch\_size:

if self.state.train\_batch\_size != self.\_train\_batch\_size:

from accelerate.utils import release\_memory

(self.model\_wrapped,) = release\_memory(self.model\_wrapped)

self.model\_wrapped = self.model

# Check for DeepSpeed \*after\* the initial pass and modify the config

if self.is\_deepspeed\_enabled:

# Temporarily unset `self.args.train\_batch\_size`

original\_bs = self.args.per\_device\_train\_batch\_size

self.args.per\_device\_train\_batch\_size = self.\_train\_batch\_size // max(1, self.args.n\_gpu)

self.propagate\_args\_to\_deepspeed(True)

self.args.per\_device\_train\_batch\_size = original\_bs

self.state.train\_batch\_size = self.\_train\_batch\_size

logger.debug(f"Currently training with a batch size of: {self.\_train\_batch\_size}")

# Data loader and number of training steps

train\_dataloader = self.get\_train\_dataloader()

if self.is\_fsdp\_xla\_v2\_enabled:

train\_dataloader = tpu\_spmd\_dataloader(train\_dataloader)

# Setting up training control variables:

# number of training epochs: num\_train\_epochs

# number of training steps per epoch: num\_update\_steps\_per\_epoch

# total number of training steps to execute: max\_steps

total\_train\_batch\_size = self.\_train\_batch\_size \* args.gradient\_accumulation\_steps \* args.world\_size

(

num\_train\_epochs,

num\_update\_steps\_per\_epoch,

num\_examples,

num\_train\_samples,

epoch\_based,

len\_dataloader,

max\_steps,

) = self.set\_initial\_training\_values(args, train\_dataloader, total\_train\_batch\_size)

num\_train\_tokens = None

if self.args.include\_tokens\_per\_second:

num\_train\_tokens = self.num\_tokens(train\_dataloader, None if epoch\_based else max\_steps)

# If going by epochs, multiply tokens linearly

if len\_dataloader is not None and epoch\_based:

num\_train\_tokens \*= args.num\_train\_epochs

# Otherwise since its steps, we just multiply by grad accum

else:

num\_train\_tokens \*= args.gradient\_accumulation\_steps

if DebugOption.UNDERFLOW\_OVERFLOW in self.args.debug:

if self.args.n\_gpu > 1:

# nn.DataParallel(model) replicates the model, creating new variables and module

# references registered here no longer work on other gpus, breaking the module

raise ValueError(

"Currently --debug underflow\_overflow is not supported under DP. Please use DDP"

" (torchrun or torch.distributed.launch (deprecated))."

)

else:

debug\_overflow = DebugUnderflowOverflow(self.model) # noqa

delay\_optimizer\_creation = is\_sagemaker\_mp\_enabled() or self.is\_fsdp\_xla\_enabled or self.is\_fsdp\_enabled

# Can't delay optimizer creation when using FSDP2: https://github.com/huggingface/accelerate/blob/3f636d626063ffcf9a337c7d3624d61b7d187d59/src/accelerate/accelerator.py#L1404

is\_fsdp2 = self.is\_fsdp\_enabled and (getattr(self.accelerator.state.fsdp\_plugin, "fsdp\_version", 1) == 2)

if is\_fsdp2:

delay\_optimizer\_creation = False

# We need to reset the scheduler, as its parameters may be different on subsequent calls

if self.\_created\_lr\_scheduler:

self.lr\_scheduler = None

self.\_created\_lr\_scheduler = False

if self.is\_deepspeed\_enabled:

self.optimizer, self.lr\_scheduler = deepspeed\_init(self, num\_training\_steps=max\_steps)

if not delay\_optimizer\_creation:

self.create\_optimizer\_and\_scheduler(num\_training\_steps=max\_steps)

self.state = TrainerState(

stateful\_callbacks=[

cb for cb in self.callback\_handler.callbacks + [self.control] if isinstance(cb, ExportableState)

]

)

self.state.is\_hyper\_param\_search = trial is not None

self.state.train\_batch\_size = self.\_train\_batch\_size

# Compute absolute values for logging, eval, and save if given as ratio

self.state.compute\_steps(args, max\_steps)

# Activate gradient checkpointing if needed

if args.gradient\_checkpointing:

self.model.gradient\_checkpointing\_enable(gradient\_checkpointing\_kwargs=args.gradient\_checkpointing\_kwargs)

model = self.\_wrap\_model(self.model\_wrapped)

# as the model is wrapped, don't use `accelerator.prepare`

# this is for unhandled cases such as

# FSDP-XLA, SageMaker MP/DP, DataParallel, IPEX

use\_accelerator\_prepare = True if model is self.model else False

if use\_accelerator\_prepare and self.is\_fsdp\_enabled:

# In case of auto\_find\_batch\_size=True

# Remove FSDP wrapping from sub-models.

self.model = unwrap\_model(self.model, recursive=True)

if delay\_optimizer\_creation:

if use\_accelerator\_prepare:

# configure fsdp plugin for qlora if any

self.\_fsdp\_qlora\_plugin\_updates()

if self.accelerator.mixed\_precision != "fp8":

self.model = self.accelerator.prepare(self.model)

self.create\_optimizer\_and\_scheduler(num\_training\_steps=max\_steps)

# prepare using `accelerator` prepare

if use\_accelerator\_prepare:

self.model.train()

if hasattr(self.lr\_scheduler, "step"):

if self.use\_apex:

model = self.accelerator.prepare(self.model)

else:

model, self.optimizer = self.accelerator.prepare(self.model, self.optimizer)

else:

# to handle cases wherein we pass "DummyScheduler" such as when it is specified in DeepSpeed config.

model, self.optimizer, self.lr\_scheduler = self.accelerator.prepare(

self.model, self.optimizer, self.lr\_scheduler

)

elif self.args.optim in [OptimizerNames.LOMO, OptimizerNames.ADALOMO]:

# In this case we are in DDP + LOMO, which should be supported

self.optimizer = self.accelerator.prepare(self.optimizer)

if self.is\_fsdp\_enabled:

self.model = self.model\_wrapped = model

# for the rest of this function `model` is the outside model, whether it was wrapped or not

if model is not self.model:

self.model\_wrapped = model

# backward compatibility

if self.is\_deepspeed\_enabled:

self.deepspeed = self.model\_wrapped

# ckpt loading

if resume\_from\_checkpoint is not None:

if self.is\_deepspeed\_enabled:

deepspeed\_load\_checkpoint(

self.model\_wrapped, resume\_from\_checkpoint, load\_module\_strict=not \_is\_peft\_model(self.model)

)

elif is\_sagemaker\_mp\_enabled() or self.is\_fsdp\_enabled:

self.\_load\_from\_checkpoint(resume\_from\_checkpoint, self.model\_wrapped)

# Check if saved optimizer or scheduler states exist

self.\_load\_optimizer\_and\_scheduler(resume\_from\_checkpoint)

self.\_load\_scaler(resume\_from\_checkpoint)

# important: at this point:

# self.model is the Transformers Model

# self.model\_wrapped is DDP(Transformers Model), Deepspeed(Transformers Model),

# FSDP(Transformers Model), Dynamo Optimized Module(Transformers Model) etc.

# Train!

logger.info("\*\*\*\*\* Running training \*\*\*\*\*")

logger.info(f" Num examples = {num\_examples:,}")

logger.info(f" Num Epochs = {num\_train\_epochs:,}")

logger.info(f" Instantaneous batch size per device = {self.args.per\_device\_train\_batch\_size:,}")

if self.args.per\_device\_train\_batch\_size != self.\_train\_batch\_size:

logger.info(f" Training with DataParallel so batch size has been adjusted to: {self.\_train\_batch\_size:,}")

logger.info(f" Total train batch size (w. parallel, distributed & accumulation) = {total\_train\_batch\_size:,}")

logger.info(f" Gradient Accumulation steps = {args.gradient\_accumulation\_steps}")

logger.info(f" Total optimization steps = {max\_steps:,}")

logger.info(f" Number of trainable parameters = {get\_model\_param\_count(model, trainable\_only=True):,}")

self.state.epoch = 0

start\_time = time.time()

epochs\_trained = 0

steps\_trained\_in\_current\_epoch = 0

steps\_trained\_progress\_bar = None

# Check if continuing training from a checkpoint

if resume\_from\_checkpoint is not None and os.path.isfile(

os.path.join(resume\_from\_checkpoint, TRAINER\_STATE\_NAME)

):

self.state = TrainerState.load\_from\_json(os.path.join(resume\_from\_checkpoint, TRAINER\_STATE\_NAME))

self.compare\_trainer\_and\_checkpoint\_args(self.args, self.state)

self.\_load\_callback\_state()

epochs\_trained = int(self.state.global\_step // num\_update\_steps\_per\_epoch)

if not args.ignore\_data\_skip:

steps\_trained\_in\_current\_epoch = self.state.global\_step % (num\_update\_steps\_per\_epoch)

steps\_trained\_in\_current\_epoch \*= args.gradient\_accumulation\_steps

else:

steps\_trained\_in\_current\_epoch = 0

logger.info(" Continuing training from checkpoint, will skip to saved global\_step")

logger.info(f" Continuing training from epoch {epochs\_trained}")

logger.info(f" Continuing training from global step {self.state.global\_step}")

if not args.ignore\_data\_skip:

logger.info(

f" Will skip the first {epochs\_trained} epochs then the first"

f" {steps\_trained\_in\_current\_epoch} batches in the first epoch."

)

# Update the references

for attr in ("model", "optimizer", "lr\_scheduler"):

setattr(self.callback\_handler, attr, getattr(self, attr))

self.callback\_handler.train\_dataloader = train\_dataloader

self.state.init\_training\_references(self, max\_steps, num\_train\_epochs, trial)

# tr\_loss is a tensor to avoid synchronization of TPUs through .item()

tr\_loss = torch.tensor(0.0, device=args.device)

# \_total\_loss\_scalar is updated everytime .item() has to be called on tr\_loss and stores the sum of all losses

self.\_total\_loss\_scalar = 0.0

self.\_globalstep\_last\_logged = self.state.global\_step

model.zero\_grad()

grad\_norm: Optional[float] = None

learning\_rate = None

self.control = self.callback\_handler.on\_train\_begin(args, self.state, self.control)

if args.eval\_on\_start:

self.\_evaluate(trial, ignore\_keys\_for\_eval, skip\_scheduler=True)

for epoch in range(epochs\_trained, num\_train\_epochs):

epoch\_dataloader = train\_dataloader

if hasattr(epoch\_dataloader, "set\_epoch"):

epoch\_dataloader.set\_epoch(epoch)

# Reset the past mems state at the beginning of each epoch if necessary.

if args.past\_index >= 0:

self.\_past = None

steps\_in\_epoch = (

len(epoch\_dataloader)

if len\_dataloader is not None

else args.max\_steps \* args.gradient\_accumulation\_steps

)

self.control = self.callback\_handler.on\_epoch\_begin(args, self.state, self.control)

if epoch == epochs\_trained and resume\_from\_checkpoint is not None and steps\_trained\_in\_current\_epoch == 0:

self.\_load\_rng\_state(resume\_from\_checkpoint)

rng\_to\_sync = False

steps\_skipped = 0

if steps\_trained\_in\_current\_epoch > 0:

epoch\_dataloader = skip\_first\_batches(epoch\_dataloader, steps\_trained\_in\_current\_epoch)

steps\_skipped = steps\_trained\_in\_current\_epoch

steps\_trained\_in\_current\_epoch = 0

rng\_to\_sync = True

step = -1

epoch\_iterator = iter(epoch\_dataloader)

# We chunkify the epoch iterator into gradient accumulation steps `n` batches

remainder = num\_examples % args.gradient\_accumulation\_steps

if remainder == 0:

remainder = args.gradient\_accumulation\_steps

update\_step = -1

total\_updates = steps\_in\_epoch // args.gradient\_accumulation\_steps + 1

if args.gradient\_accumulation\_steps == 1:

total\_updates -= 1

for \_ in range(total\_updates):

update\_step += 1

num\_batches = args.gradient\_accumulation\_steps if update\_step != (total\_updates - 1) else remainder

batch\_samples, num\_items\_in\_batch = self.get\_batch\_samples(epoch\_iterator, num\_batches, args.device)

for i, inputs in enumerate(batch\_samples):

step += 1

do\_sync\_step = (step + 1) % args.gradient\_accumulation\_steps == 0 or (step + 1) == steps\_in\_epoch

# Since we perform prefetching, we need to manually set sync\_gradients

self.accelerator.gradient\_state.\_set\_sync\_gradients(do\_sync\_step)

if self.args.include\_num\_input\_tokens\_seen:

main\_input\_name = getattr(self.model, "main\_input\_name", "input\_ids")

if main\_input\_name not in inputs:

logger.warning(

"Tried to track the number of tokens seen, however the current model is "

"not configured properly to know what item is the input. To fix this, add "

"a `main\_input\_name` attribute to the model class you are using."

)

else:

input\_tokens = inputs[main\_input\_name].numel()

input\_tokens = torch.tensor(input\_tokens, device=self.args.device, dtype=torch.int64)

self.state.num\_input\_tokens\_seen += self.accelerator.gather(input\_tokens).sum().item()

if rng\_to\_sync:

self.\_load\_rng\_state(resume\_from\_checkpoint)

rng\_to\_sync = False

# Skip past any already trained steps if resuming training

if steps\_trained\_in\_current\_epoch > 0:

steps\_trained\_in\_current\_epoch -= 1

if steps\_trained\_progress\_bar is not None:

steps\_trained\_progress\_bar.update(1)

if steps\_trained\_in\_current\_epoch == 0:

self.\_load\_rng\_state(resume\_from\_checkpoint)

continue

elif steps\_trained\_progress\_bar is not None:

steps\_trained\_progress\_bar.close()

steps\_trained\_progress\_bar = None

if step % args.gradient\_accumulation\_steps == 0:

self.control = self.callback\_handler.on\_step\_begin(args, self.state, self.control)

# We explicitly want to avoid relying on `accelerator.accumulate` for generation training

context = (

functools.partial(self.accelerator.no\_sync, model=model)

if i != len(batch\_samples) - 1

and self.accelerator.distributed\_type != DistributedType.DEEPSPEED

else contextlib.nullcontext

)

with context():

tr\_loss\_step = self.training\_step(model, inputs, num\_items\_in\_batch)

if (

args.logging\_nan\_inf\_filter

and not is\_torch\_xla\_available()

and (torch.isnan(tr\_loss\_step) or torch.isinf(tr\_loss\_step))

):

# if loss is nan or inf simply add the average of previous logged losses

tr\_loss = tr\_loss + tr\_loss / (1 + self.state.global\_step - self.\_globalstep\_last\_logged)

else:

if tr\_loss.device != tr\_loss\_step.device:

raise ValueError(

f"Calculated loss must be on the original device: {tr\_loss.device} but device in use is {tr\_loss\_step.device}"

)

tr\_loss = tr\_loss + tr\_loss\_step

self.current\_flos += float(self.floating\_point\_ops(inputs))

if do\_sync\_step:

# Since we perform prefetching, we need to manually set sync\_gradients to True

self.accelerator.gradient\_state.\_set\_sync\_gradients(True)

# Gradient clipping

if args.max\_grad\_norm is not None and args.max\_grad\_norm > 0:

if is\_sagemaker\_mp\_enabled() and args.fp16:

\_grad\_norm = self.optimizer.clip\_master\_grads(args.max\_grad\_norm)

elif self.use\_apex:

# Revert to normal clipping otherwise, handling Apex or full precision

\_grad\_norm = nn.utils.clip\_grad\_norm\_(

amp.master\_params(self.optimizer),

args.max\_grad\_norm,

)

else:

\_grad\_norm = self.accelerator.clip\_grad\_norm\_(

model.parameters(),

args.max\_grad\_norm,

)

if (

is\_accelerate\_available()

and self.accelerator.distributed\_type == DistributedType.DEEPSPEED

):

grad\_norm = model.get\_global\_grad\_norm()

# In some cases the grad norm may not return a float

if hasattr(grad\_norm, "item"):

grad\_norm = grad\_norm.item()

else:

grad\_norm = \_grad\_norm

self.control = self.callback\_handler.on\_pre\_optimizer\_step(args, self.state, self.control)

self.optimizer.step()

self.control = self.callback\_handler.on\_optimizer\_step(args, self.state, self.control)

# get leaning rate before update

learning\_rate = self.\_get\_learning\_rate()

if not self.accelerator.optimizer\_step\_was\_skipped:

# Delay optimizer scheduling until metrics are generated

if not isinstance(self.lr\_scheduler, torch.optim.lr\_scheduler.ReduceLROnPlateau):

self.lr\_scheduler.step()

model.zero\_grad()

self.state.global\_step += 1

self.state.epoch = epoch + (step + 1 + steps\_skipped) / steps\_in\_epoch

self.control = self.callback\_handler.on\_step\_end(args, self.state, self.control)

self.\_maybe\_log\_save\_evaluate(

tr\_loss,

grad\_norm,

model,

trial,

epoch,

ignore\_keys\_for\_eval,

start\_time,

learning\_rate=learning\_rate,

)

else:

self.control = self.callback\_handler.on\_substep\_end(args, self.state, self.control)

# PyTorch/XLA relies on the data loader to insert the mark\_step for

# each step. Since we are breaking the loop early, we need to manually

# insert the mark\_step here.

if self.control.should\_epoch\_stop or self.control.should\_training\_stop:

if is\_torch\_xla\_available():

xm.mark\_step()

break

# We also need to break out of the nested loop

if self.control.should\_epoch\_stop or self.control.should\_training\_stop:

if is\_torch\_xla\_available():

xm.mark\_step()

break

if step < 0:

logger.warning(

"There seems not to be a single sample in your epoch\_iterator, stopping training at step"

f" {self.state.global\_step}! This is expected if you're using an IterableDataset and set"

f" num\_steps ({max\_steps}) higher than the number of available samples."

)

self.control.should\_training\_stop = True

self.control = self.callback\_handler.on\_epoch\_end(args, self.state, self.control)

self.\_maybe\_log\_save\_evaluate(

tr\_loss, grad\_norm, model, trial, epoch, ignore\_keys\_for\_eval, start\_time, learning\_rate=learning\_rate

)

if DebugOption.TPU\_METRICS\_DEBUG in self.args.debug:

if is\_torch\_xla\_available():

# tpu-comment: Logging debug metrics for PyTorch/XLA (compile, execute times, ops, etc.)

xm.master\_print(met.metrics\_report())

else:

logger.warning(

"You enabled PyTorch/XLA debug metrics but you don't have a TPU "

"configured. Check your training configuration if this is unexpected."

)

if self.control.should\_training\_stop:

break

if args.past\_index and hasattr(self, "\_past"):

# Clean the state at the end of training

delattr(self, "\_past")

logger.info("\n\nTraining completed. Do not forget to share your model on huggingface.co/models =)\n\n")

if args.load\_best\_model\_at\_end and self.state.best\_model\_checkpoint is not None:

# Wait for everyone to get here so we are sure the model has been saved by process 0.

if is\_torch\_xla\_available():

xm.rendezvous("load\_best\_model\_at\_end")

elif args.parallel\_mode == ParallelMode.DISTRIBUTED:

dist.barrier()

elif is\_sagemaker\_mp\_enabled():

smp.barrier()

self.\_load\_best\_model()

# add remaining tr\_loss

self.\_total\_loss\_scalar += tr\_loss.item()

effective\_global\_step = max(self.state.global\_step, 0.001) # Avoid ZeroDivisionError

train\_loss = self.\_total\_loss\_scalar / effective\_global\_step

metrics = speed\_metrics(

"train",

start\_time,

num\_samples=num\_train\_samples,

num\_steps=self.state.max\_steps,

num\_tokens=num\_train\_tokens,

)

self.store\_flos()

metrics["total\_flos"] = self.state.total\_flos

metrics["train\_loss"] = train\_loss

self.is\_in\_train = False

self.\_memory\_tracker.stop\_and\_update\_metrics(metrics)

self.log(metrics)

run\_dir = self.\_get\_output\_dir(trial)

checkpoints\_sorted = self.\_sorted\_checkpoints(use\_mtime=False, output\_dir=run\_dir)

# Delete the last checkpoint when save\_total\_limit=1 if it's different from the best checkpoint and process allowed to save.

if self.args.should\_save and self.state.best\_model\_checkpoint is not None and self.args.save\_total\_limit == 1:

for checkpoint in checkpoints\_sorted:

if not os.path.samefile(checkpoint, self.state.best\_model\_checkpoint):

logger.info(f"Deleting older checkpoint [{checkpoint}] due to args.save\_total\_limit")

shutil.rmtree(checkpoint, ignore\_errors=True)

self.control = self.callback\_handler.on\_train\_end(args, self.state, self.control)

# Wait for the checkpoint to be uploaded.

self.\_finish\_current\_push()

# After training we make sure to retrieve back the original forward pass method

# for the embedding layer by removing the forward post hook.

if self.neftune\_noise\_alpha is not None:

self.\_deactivate\_neftune(self.model)

return TrainOutput(self.state.global\_step, train\_loss, metrics)

### Trainer::training\_step

<https://github.com/huggingface/transformers/blob/main/src/transformers/trainer.py#L3698-L3776>

def training\_step(

self, model: nn.Module, inputs: dict[str, Union[torch.Tensor, Any]], num\_items\_in\_batch=None

) -> torch.Tensor:

"""

Perform a training step on a batch of inputs.

Subclass and override to inject custom behavior.

Args:

model (`nn.Module`):

The model to train.

inputs (`Dict[str, Union[torch.Tensor, Any]]`):

The inputs and targets of the model.

The dictionary will be unpacked before being fed to the model. Most models expect the targets under the

argument `labels`. Check your model's documentation for all accepted arguments.

Return:

`torch.Tensor`: The tensor with training loss on this batch.

"""

model.train()

if hasattr(self.optimizer, "train") and callable(self.optimizer.train):

self.optimizer.train()

inputs = self.\_prepare\_inputs(inputs)

if is\_sagemaker\_mp\_enabled():

loss\_mb = smp\_forward\_backward(model, inputs, self.args.gradient\_accumulation\_steps)

return loss\_mb.reduce\_mean().detach().to(self.args.device)

with self.compute\_loss\_context\_manager():

loss = self.compute\_loss(model, inputs, num\_items\_in\_batch=num\_items\_in\_batch)

del inputs

if (

self.args.torch\_empty\_cache\_steps is not None

and self.state.global\_step % self.args.torch\_empty\_cache\_steps == 0

):

if is\_torch\_xpu\_available():

torch.xpu.empty\_cache()

elif is\_torch\_mlu\_available():

torch.mlu.empty\_cache()

elif is\_torch\_musa\_available():

torch.musa.empty\_cache()

elif is\_torch\_npu\_available():

torch.npu.empty\_cache()

elif is\_torch\_mps\_available():

torch.mps.empty\_cache()

elif is\_torch\_hpu\_available():

logger.warning(

"`torch\_empty\_cache\_steps` is set but HPU device/backend does not support empty\_cache()."

)

else:

torch.cuda.empty\_cache()

kwargs = {}

# For LOMO optimizers you need to explicitly use the learnign rate

if self.args.optim in [OptimizerNames.LOMO, OptimizerNames.ADALOMO]:

kwargs["learning\_rate"] = self.\_get\_learning\_rate()

if self.args.n\_gpu > 1:

loss = loss.mean() # mean() to average on multi-gpu parallel training

if self.use\_apex:

with amp.scale\_loss(loss, self.optimizer) as scaled\_loss:

scaled\_loss.backward()

else:

# Finally we need to normalize the loss for reporting

if not self.model\_accepts\_loss\_kwargs and self.compute\_loss\_func is None:

loss = loss / self.args.gradient\_accumulation\_steps

# Turning off loss scaling w.r.t. gradient accumulation when DeepSpeed is enabled

# https://github.com/huggingface/transformers/pull/35808

if self.accelerator.distributed\_type == DistributedType.DEEPSPEED:

kwargs["scale\_wrt\_gas"] = False

self.accelerator.backward(loss, \*\*kwargs)

return loss.detach()

### Trainer::compute\_loss

<https://github.com/huggingface/transformers/blob/main/src/transformers/trainer.py#L3778-L3828>

def compute\_loss(self, model, inputs, return\_outputs=False, num\_items\_in\_batch=None):

"""

How the loss is computed by Trainer. By default, all models return the loss in the first element.

Subclass and override for custom behavior.

"""

if (self.label\_smoother is not None or self.compute\_loss\_func is not None) and "labels" in inputs:

labels = inputs.pop("labels")

else:

labels = None

if self.model\_accepts\_loss\_kwargs:

loss\_kwargs = {}

if num\_items\_in\_batch is not None:

loss\_kwargs["num\_items\_in\_batch"] = num\_items\_in\_batch

inputs = {\*\*inputs, \*\*loss\_kwargs}

outputs = model(\*\*inputs)

# Save past state if it exists

# TODO: this needs to be fixed and made cleaner later.

if self.args.past\_index >= 0:

self.\_past = outputs[self.args.past\_index]

if labels is not None:

unwrapped\_model = self.accelerator.unwrap\_model(model)

if \_is\_peft\_model(unwrapped\_model):

model\_name = unwrapped\_model.base\_model.model.\_get\_name()

else:

model\_name = unwrapped\_model.\_get\_name()

# User-defined compute\_loss function

if self.compute\_loss\_func is not None:

loss = self.compute\_loss\_func(outputs, labels, num\_items\_in\_batch=num\_items\_in\_batch)

elif model\_name in MODEL\_FOR\_CAUSAL\_LM\_MAPPING\_NAMES.values():

loss = self.label\_smoother(outputs, labels, shift\_labels=True)

else:

loss = self.label\_smoother(outputs, labels)

else:

if isinstance(outputs, dict) and "loss" not in outputs:

raise ValueError(

"The model did not return a loss from the inputs, only the following keys: "

f"{','.join(outputs.keys())}. For reference, the inputs it received are {','.join(inputs.keys())}."

)

# We don't use .loss here since the model may return tuples instead of ModelOutput.

loss = outputs["loss"] if isinstance(outputs, dict) else outputs[0]

if (

self.args.average\_tokens\_across\_devices

and (self.model\_accepts\_loss\_kwargs or self.compute\_loss\_func)

and num\_items\_in\_batch is not None

):

loss \*= self.accelerator.num\_processes

return (loss, outputs) if return\_outputs else loss

## Loss Functions in Torch

### Cross Entropy

<https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py#L3384-L3474>

For applicability see HF discussion [here](https://discuss.huggingface.co/t/which-loss-function-in-bertforsequenceclassification-regression/1432).

def cross\_entropy(

input: Tensor,

target: Tensor,

weight: Optional[Tensor] = None,

size\_average: Optional[bool] = None,

ignore\_index: int = -100,

reduce: Optional[bool] = None,

reduction: str = "mean",

label\_smoothing: float = 0.0,

) -> Tensor:

r"""Compute the cross entropy loss between input logits and target.

See :class:`~torch.nn.CrossEntropyLoss` for details.

Args:

input (Tensor) : Predicted unnormalized logits;

see Shape section below for supported shapes.

target (Tensor) : Ground truth class indices or class probabilities;

see Shape section below for supported shapes.

weight (Tensor, optional): a manual rescaling weight given to each

class. If given, has to be a Tensor of size `C`

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

ignore\_index (int, optional): Specifies a target value that is ignored

and does not contribute to the input gradient. When :attr:`size\_average` is

``True``, the loss is averaged over non-ignored targets. Note that

:attr:`ignore\_index` is only applicable when the target contains class indices.

Default: -100

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

``'none'`` | ``'mean'`` | ``'sum'``. ``'none'``: no reduction will be applied,

``'mean'``: the sum of the output will be divided by the number of

elements in the output, ``'sum'``: the output will be summed. Note: :attr:`size\_average`

and :attr:`reduce` are in the process of being deprecated, and in the meantime,

specifying either of those two args will override :attr:`reduction`. Default: ``'mean'``

label\_smoothing (float, optional): A float in [0.0, 1.0]. Specifies the amount

of smoothing when computing the loss, where 0.0 means no smoothing. The targets

become a mixture of the original ground truth and a uniform distribution as described in

`Rethinking the Inception Architecture for Computer Vision <https://arxiv.org/abs/1512.00567>`\_\_. Default: :math:`0.0`.

Shape:

- Input: Shape :math:`(C)`, :math:`(N, C)` or :math:`(N, C, d\_1, d\_2, ..., d\_K)` with :math:`K \geq 1`

in the case of `K`-dimensional loss.

- Target: If containing class indices, shape :math:`()`, :math:`(N)` or :math:`(N, d\_1, d\_2, ..., d\_K)` with

:math:`K \geq 1` in the case of K-dimensional loss where each value should be between :math:`[0, C)`.

If containing class probabilities, same shape as the input and each value should be between :math:`[0, 1]`.

where:

.. math::

\begin{aligned}

C ={} & \text{number of classes} \\

N ={} & \text{batch size} \\

\end{aligned}

Examples::

>>> # Example of target with class indices

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> target = torch.randint(5, (3,), dtype=torch.int64)

>>> loss = F.cross\_entropy(input, target)

>>> loss.backward()

>>>

>>> # Example of target with class probabilities

>>> input = torch.randn(3, 5, requires\_grad=True)

>>> target = torch.randn(3, 5).softmax(dim=1)

>>> loss = F.cross\_entropy(input, target)

>>> loss.backward()

"""

if has\_torch\_function\_variadic(input, target, weight):

return handle\_torch\_function(

cross\_entropy,

(input, target, weight),

input,

target,

weight=weight,

size\_average=size\_average,

ignore\_index=ignore\_index,

reduce=reduce,

reduction=reduction,

label\_smoothing=label\_smoothing,

)

if size\_average is not None or reduce is not None:

reduction = \_Reduction.legacy\_get\_string(size\_average, reduce)

return torch.\_C.\_nn.cross\_entropy\_loss(

input,

target,

weight,

\_Reduction.get\_enum(reduction),

ignore\_index,

label\_smoothing,

)

### Binary Cross Entropy

<https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py#L3477-L3535>

def binary\_cross\_entropy(

input: Tensor,

target: Tensor,

weight: Optional[Tensor] = None,

size\_average: Optional[bool] = None,

reduce: Optional[bool] = None,

reduction: str = "mean",

) -> Tensor:

r"""Compute Binary Cross Entropy between the target and input probabilities.

See :class:`~torch.nn.BCELoss` for details.

Args:

input: Tensor of arbitrary shape as probabilities.

target: Tensor of the same shape as input with values between 0 and 1.

weight (Tensor, optional): a manual rescaling weight

if provided it's repeated to match input tensor shape

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

``'none'`` | ``'mean'`` | ``'sum'``. ``'none'``: no reduction will be applied,

``'mean'``: the sum of the output will be divided by the number of

elements in the output, ``'sum'``: the output will be summed. Note: :attr:`size\_average`

and :attr:`reduce` are in the process of being deprecated, and in the meantime,

specifying either of those two args will override :attr:`reduction`. Default: ``'mean'``

Examples::

>>> input = torch.randn(3, 2, requires\_grad=True)

>>> target = torch.rand(3, 2, requires\_grad=False)

>>> loss = F.binary\_cross\_entropy(torch.sigmoid(input), target)

>>> loss.backward()

"""

if has\_torch\_function\_variadic(input, target, weight):

return handle\_torch\_function(

binary\_cross\_entropy,

(input, target, weight),

input,

target,

weight=weight,

size\_average=size\_average,

reduce=reduce,

reduction=reduction,

)

if size\_average is not None or reduce is not None:

reduction\_enum = \_Reduction.legacy\_get\_enum(size\_average, reduce)

else:

reduction\_enum = \_Reduction.get\_enum(reduction)

if target.size() != input.size():

raise ValueError(

f"Using a target size ({target.size()}) that is different to the input size ({input.size()}) is deprecated. "

"Please ensure they have the same size."

)

if weight is not None:

new\_size = \_infer\_size(target.size(), weight.size())

weight = weight.expand(new\_size)

return torch.\_C.\_nn.binary\_cross\_entropy(input, target, weight, reduction\_enum)

### Binary Cross Entropy With Logits

<https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py#L3538-L3604>

def binary\_cross\_entropy\_with\_logits(

input: Tensor,

target: Tensor,

weight: Optional[Tensor] = None,

size\_average: Optional[bool] = None,

reduce: Optional[bool] = None,

reduction: str = "mean",

pos\_weight: Optional[Tensor] = None,

) -> Tensor:

r"""Compute Binary Cross Entropy between target and input logits.

See :class:`~torch.nn.BCEWithLogitsLoss` for details.

Args:

input: Tensor of arbitrary shape as unnormalized scores (often referred to as logits).

target: Tensor of the same shape as input with values between 0 and 1

weight (Tensor, optional): a manual rescaling weight

if provided it's repeated to match input tensor shape

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

``'none'`` | ``'mean'`` | ``'sum'``. ``'none'``: no reduction will be applied,

``'mean'``: the sum of the output will be divided by the number of

elements in the output, ``'sum'``: the output will be summed. Note: :attr:`size\_average`

and :attr:`reduce` are in the process of being deprecated, and in the meantime,

specifying either of those two args will override :attr:`reduction`. Default: ``'mean'``

pos\_weight (Tensor, optional): a weight of positive examples to be broadcasted with target.

Must be a tensor with equal size along the class dimension to the number of classes.

Pay close attention to PyTorch's broadcasting semantics in order to achieve the desired

operations. For a target of size [B, C, H, W] (where B is batch size) pos\_weight of

size [B, C, H, W] will apply different pos\_weights to each element of the batch or

[C, H, W] the same pos\_weights across the batch. To apply the same positive weight

along all spatial dimensions for a 2D multi-class target [C, H, W] use: [C, 1, 1].

Default: ``None``

Examples::

>>> input = torch.randn(3, requires\_grad=True)

>>> target = torch.empty(3).random\_(2)

>>> loss = F.binary\_cross\_entropy\_with\_logits(input, target)

>>> loss.backward()

"""

if has\_torch\_function\_variadic(input, target, weight, pos\_weight):

return handle\_torch\_function(

binary\_cross\_entropy\_with\_logits,

(input, target, weight, pos\_weight),

input,

target,

weight=weight,

size\_average=size\_average,

reduce=reduce,

reduction=reduction,

pos\_weight=pos\_weight,

)

if size\_average is not None or reduce is not None:

reduction\_enum = \_Reduction.legacy\_get\_enum(size\_average, reduce)

else:

reduction\_enum = \_Reduction.get\_enum(reduction)

if not (target.size() == input.size()):

raise ValueError(

f"Target size ({target.size()}) must be the same as input size ({input.size()})"

)

return torch.binary\_cross\_entropy\_with\_logits(

input, target, weight, pos\_weight, reduction\_enum

)

### Smooth L1 Loss

def smooth\_l1\_loss(

input: Tensor,

target: Tensor,

size\_average: Optional[bool] = None,

reduce: Optional[bool] = None,

reduction: str = "mean",

beta: float = 1.0,

) -> Tensor:

r"""Compute the Smooth L1 loss.

Function uses a squared term if the absolute

element-wise error falls below beta and an L1 term otherwise.

See :class:`~torch.nn.SmoothL1Loss` for details.

Args:

input (Tensor): Predicted values.

target (Tensor): Ground truth values.

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

'none' | 'mean' | 'sum'. 'mean': the mean of the output is taken.

'sum': the output will be summed. 'none': no reduction will be applied.

Default: 'mean'.

beta (float, optional): Specifies the threshold at which to change from the squared

term to the L1 term in the loss calculation. This value must be positive.

Default: 1.0.

Returns:

Tensor: L1 loss (optionally weighted).

"""

if has\_torch\_function\_variadic(input, target):

return handle\_torch\_function(

smooth\_l1\_loss,

(input, target),

input,

target,

size\_average=size\_average,

reduce=reduce,

reduction=reduction,

beta=beta,

)

if not (target.size() == input.size()):

warnings.warn(

f"Using a target size ({target.size()}) that is different to the input size ({input.size()}). "

"This will likely lead to incorrect results due to broadcasting. "

"Please ensure they have the same size.",

stacklevel=2,

)

if size\_average is not None or reduce is not None:

reduction = \_Reduction.legacy\_get\_string(size\_average, reduce)

expanded\_input, expanded\_target = torch.broadcast\_tensors(input, target)

if beta == 0.0:

return torch.\_C.\_nn.l1\_loss(

expanded\_input, expanded\_target, \_Reduction.get\_enum(reduction)

)

else:

return torch.\_C.\_nn.smooth\_l1\_loss(

expanded\_input, expanded\_target, \_Reduction.get\_enum(reduction), beta

)

### L1 loss

<https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py#L3751-L3821>

def l1\_loss(

input: Tensor,

target: Tensor,

size\_average: Optional[bool] = None,

reduce: Optional[bool] = None,

reduction: str = "mean",

weight: Optional[Tensor] = None,

) -> Tensor: # noqa: D400,D402

r"""Compute the L1 loss, with optional weighting.

Function that takes the mean element-wise absolute value difference.

See :class:`~torch.nn.L1Loss` for details.

Args:

input (Tensor): Predicted values.

target (Tensor): Ground truth values.

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

'none' | 'mean' | 'sum'. 'mean': the mean of the output is taken.

'sum': the output will be summed. 'none': no reduction will be applied.

Default: 'mean'.

weight (Tensor, optional): Weights for each sample. Default: None.

Returns:

Tensor: L1 loss (optionally weighted).

"""

if has\_torch\_function\_variadic(input, target):

return handle\_torch\_function(

l1\_loss,

(input, target, weight),

input,

target,

size\_average=size\_average,

reduce=reduce,

reduction=reduction,

)

if not (target.size() == input.size()):

warnings.warn(

f"Using a target size ({target.size()}) that is different to the input size ({input.size()}). "

"This will likely lead to incorrect results due to broadcasting. "

"Please ensure they have the same size.",

stacklevel=2,

)

if size\_average is not None or reduce is not None:

reduction = \_Reduction.legacy\_get\_string(size\_average, reduce)

expanded\_input, expanded\_target = torch.broadcast\_tensors(input, target)

if weight is not None:

if weight.size() != input.size():

raise ValueError("Weights and input must have the same size.")

absolute\_errors = torch.abs(expanded\_input - expanded\_target)

weighted\_absolute\_errors = absolute\_errors \* weight

if reduction == "none":

return weighted\_absolute\_errors

elif reduction == "sum":

return torch.sum(weighted\_absolute\_errors)

elif reduction == "mean":

return torch.sum(weighted\_absolute\_errors) / torch.sum(weight)

else:

raise ValueError(

f"Invalid reduction mode: {reduction}. Expected one of 'none', 'mean', 'sum'."

)

else:

return torch.\_C.\_nn.l1\_loss(

expanded\_input, expanded\_target, \_Reduction.get\_enum(reduction)

)

### MSE Loss

<https://github.com/pytorch/pytorch/blob/main/torch/nn/functional.py#L3824-L3896>

For discussion on the applicability of the MSE loss see this HF discussion [here](https://discuss.huggingface.co/t/which-loss-function-in-bertforsequenceclassification-regression/1432).

def mse\_loss(

input: Tensor,

target: Tensor,

size\_average: Optional[bool] = None,

reduce: Optional[bool] = None,

reduction: str = "mean",

weight: Optional[Tensor] = None,

) -> Tensor:

r"""Compute the element-wise mean squared error, with optional weighting.

See :class:`~torch.nn.MSELoss` for details.

Args:

input (Tensor): Predicted values.

target (Tensor): Ground truth values.

size\_average (bool, optional): Deprecated (see :attr:`reduction`).

reduce (bool, optional): Deprecated (see :attr:`reduction`).

reduction (str, optional): Specifies the reduction to apply to the output:

'none' | 'mean' | 'sum'. 'mean': the mean of the output is taken.

'sum': the output will be summed. 'none': no reduction will be applied.

Default: 'mean'.

weight (Tensor, optional): Weights for each sample. Default: None.

Returns:

Tensor: Mean Squared Error loss (optionally weighted).

"""

if has\_torch\_function\_variadic(input, target, weight):

return handle\_torch\_function(

mse\_loss,

(input, target, weight),

input,

target,

size\_average=size\_average,

reduce=reduce,

reduction=reduction,

weight=weight,

)

if not (target.size() == input.size()):

warnings.warn(

f"Using a target size ({target.size()}) that is different to the input size ({input.size()}). "

"This will likely lead to incorrect results due to broadcasting. "

"Please ensure they have the same size.",

stacklevel=2,

)

if size\_average is not None or reduce is not None:

reduction = \_Reduction.legacy\_get\_string(size\_average, reduce)

expanded\_input, expanded\_target = torch.broadcast\_tensors(input, target)

if weight is not None:

if weight.size() != input.size():

raise ValueError("Weights and input must have the same size.")

# Perform weighted MSE loss manually

squared\_errors = torch.pow(expanded\_input - expanded\_target, 2)

weighted\_squared\_errors = squared\_errors \* weight

if reduction == "none":

return weighted\_squared\_errors

elif reduction == "sum":

return torch.sum(weighted\_squared\_errors)

elif reduction == "mean":

return torch.sum(weighted\_squared\_errors) / torch.sum(weight)

else:

raise ValueError(

f"Invalid reduction mode: {reduction}. Expected one of 'none', 'mean', 'sum'."

)

else:

return torch.\_C.\_nn.mse\_loss(

expanded\_input, expanded\_target, \_Reduction.get\_enum(reduction)

)