training

October 4, 2023

0.1 Train Dialog-Fact Encoder

Goal: Train an embedding model to match dialogs with (possibly) relevant facts

0.1.1 Constants

[1]:
```
model_name = "BAAI/bge-base-en-v1.5"
query_prefix = "Represent this sentence for searching relevant passages: "
max_len = 512
training_hn_file = "./data/train.jsonl"
eval_file = "./data/eval.jsonl"
batch_size = 1350
output_model_path = "./dfe-base-en"
hf_repo_name = "julep-ai/dfe-base-en"
```

0.1.2 Imports

[2]:
```
import itertools as it

import jsonlines as jsonl
from lion_pytorch import Lion
from sentence_transformers import InputExample, SentenceTransformer, losses as␣
 ↪ls, models as ml, util
from sentence_transformers.evaluation import SimilarityFunction,␣
 ↪TripletEvaluator
import torch
from torch.utils.data import DataLoader, IterableDataset
from tqdm.auto import tqdm
```

0.1.3 Dataset

[3]:
```
def hn_output(file):
    with jsonl.open(file) as reader:
        for entry in reader:
            query = entry["query"]
            pos = [dict(dialog=dialog) for dialog in entry["pos"]]
            neg = [dict(dialog=dialog) for dialog in entry["neg"]]
```

```
            for combined in it.product(
                [dict(fact=query)],
                pos,
                neg,
            ):
                yield InputExample(texts=list(combined))
```

[4]:
```
training_data = list(tqdm(hn_output(training_hn_file)))
eval_data = list(tqdm(hn_output(eval_file)))
```

```
0it [00:00, ?it/s]
```

```
0it [00:00, ?it/s]
```

[5]:
```
dataloader = DataLoader(training_data, shuffle=True, batch_size=batch_size)
eval_dataloader = DataLoader(eval_data, shuffle=True, batch_size=batch_size //␣
↪10)
```

0.1.4 DFE Model Architecture

[6]:
```
# Base model
base_model = SentenceTransformer(model_name)
```

[7]:
```
# Freeze base transformer layers
for param in base_model.parameters():
    param.requires_grad = False
```

[8]:
```
device = torch.device("cuda:0")

# Note that we must also set _target_device, or any SentenceTransformer.fit()␣
↪call will reset
# the body location
base_model._target_device = device
base_model = base_model.to(device)
```

[9]:
```
emb_dims = base_model._first_module().get_word_embedding_dimension() # 768

def dense_projector(dims: int):
    proj_dims = dims * 2  # 1536

    return [
        ml.Dense(dims, proj_dims),  # 768 -> 1536
        ml.Dense(proj_dims, proj_dims), # 1536 -> 1536
        ml.Dropout(0.1),
        ml.Dense(proj_dims, proj_dims), # 1536 -> 1536
        ml.Dense(proj_dims, dims),  # 1536 -> 768
    ]
```

```
def asym_module(dims: int, keys: list[str], allow_empty_key: bool = False):
    return ml.Asym(
        {
            key: dense_projector(dims)
            for key in keys
        },
        allow_empty_key=allow_empty_key,
    )
```

[10]:
```
base_model._modules["2"] = asym_module(emb_dims, ["dialog", "fact"])
```

[11]:
```
base_model._modules
```

[11]:
```
OrderedDict([('0',
              Transformer({'max_seq_length': 512, 'do_lower_case': True}) with
Transformer model: BertModel ),
             ('1',
              Pooling({'word_embedding_dimension': 768,
'pooling_mode_cls_token': True, 'pooling_mode_mean_tokens': False,
'pooling_mode_max_tokens': False, 'pooling_mode_mean_sqrt_len_tokens': False})),
             ('2',
              Asym(
                (dialog-0): Dense({'in_features': 768, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (dialog-1): Dense({'in_features': 1536, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (dialog-2): Dropout(
                  (dropout_layer): Dropout(p=0.1, inplace=False)
                )
                (dialog-3): Dense({'in_features': 1536, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (dialog-4): Dense({'in_features': 1536, 'out_features': 768,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (fact-0): Dense({'in_features': 768, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (fact-1): Dense({'in_features': 1536, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (fact-2): Dropout(
                  (dropout_layer): Dropout(p=0.1, inplace=False)
                )
                (fact-3): Dense({'in_features': 1536, 'out_features': 1536,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
                (fact-4): Dense({'in_features': 1536, 'out_features': 768,
'bias': True, 'activation_function': 'torch.nn.modules.activation.Tanh'})
              ))])
```

0.1.5 Prepare training loss and evaluator

[12]:
```
train_loss = ls.TripletLoss(model=base_model)
```

[13]:
```
triplet_evaluator = TripletEvaluator.from_input_examples(
    eval_data,  # Triplet is ({dialog: <some_dialog>}, {fact: <relevant_fact>},␣
 ↪[{fact: <negative_irrelevant_fact>}])
    batch_size=batch_size // 10,
    main_distance_function=SimilarityFunction.COSINE,
    show_progress_bar=True,
    write_csv=True,
)
```

0.1.6 Train model

[ ]:
```
base_model.fit(
    train_objectives=[(dataloader, train_loss)],
    evaluator=triplet_evaluator,
    checkpoint_save_steps=600,
    evaluation_steps=600,
    checkpoint_path=f"{output_model_path}/ckpts",
    scheduler="WarmupCosine",
    save_best_model=True,
    epochs=15,
    warmup_steps=200,
    optimizer_class=Lion,
    optimizer_params=dict(lr=1e-4, weight_decay=1e-2),
    use_amp=True,
    output_path=output_model_path,
    checkpoint_save_total_limit=4,
)
```

```
Epoch:   0%|          | 0/15 [00:00<?, ?it/s]
Iteration:   0%|          | 0/2505 [00:00<?, ?it/s]
```

[ ]:
```

```