Natural Language Inference With BERT

For this homework, we will work on (NLI)[https://nlp.stanford.edu/projects/snli/].

The task is, give two sentences: a premise and a hypothesis, to classify the relation between them. We have three classes to describe this relationship.

1. Entailment: the hypothesis follows from the fact that the premise is true
2. Contradiction: the hypothesis contradicts the fact that the premise is true
3. Neutral: There is not relationship between premise and hypothesis

See below for examples

snli_task

Prereqs

```
! pip install transformers datasets tqdm
```

```
Collecting transformers
  Downloading transformers-4.35.2-py3-none-any.whl (7.9 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.9/7.9 MB 28.9 MB/s eta 0:00:00
Collecting datasets
  Downloading datasets-2.15.0-py3-none-any.whl (521 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 521.2/521.2 kB 40.3 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (4.66.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.13.1)
Collecting huggingface-hub<1.0,>=0.16.4 (from transformers)
  Downloading huggingface_hub-0.19.4-py3-none-any.whl (311 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 311.7/311.7 kB 39.6 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
Collecting tokenizers<0.19,>=0.14 (from transformers)
  Downloading tokenizers-0.15.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (3.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.8/3.8 MB 60.9 MB/s eta 0:00:00
Collecting safetensors>=0.3.1 (from transformers)
  Downloading safetensors-0.4.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 38.9 MB/s eta 0:00:00
Requirement already satisfied: pyarrow>=8.0.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (9.0.0)
Collecting pyarrow-hotfix (from datasets)
  Downloading pyarrow_hotfix-0.5-py3-none-any.whl (7.8 kB)
Collecting dill<0.3.8,>=0.3.0 (from datasets)
  Downloading dill-0.3.7-py3-none-any.whl (115 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 115.3/115.3 kB 11.0 MB/s eta 0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from datasets) (1.5.3)
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packages (from datasets) (3.4.1)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.15-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 12.1 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]<=2023.10.0,>=2023.1.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (2023.6.0)
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-packages (from datasets) (3.8.6)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (23.1.0)
Requirement already satisfied: charset-normalizer<4.0,>=2.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (3.3.2)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (6.0.4)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (4.0.3)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.9.2)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datasets) (1.3.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.16.4->transformers) (4.5.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (2023.7.22)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas->datasets) (1.16.0)
Installing collected packages: safetensors, pyarrow-hotfix, dill, multiprocess, huggingface-hub, tokenizers, transformers, datasets
Successfully installed datasets-2.15.0 dill-0.3.7 huggingface-hub-0.19.4 multiprocess-0.70.15 pyarrow-hotfix-0.5 safetensors-0.4.0 tokenizers-0.15.0 transformers-4.35.2
```

```python
# Imports for most of the notebook
import torch
from transformers import BertModel
from transformers import AutoTokenizer
from typing import Dict, List
import random
from tqdm.autonotebook import tqdm
```

```python
print(torch.cuda.is_available())
# device = torch.device("cpu")
# TODO: Uncomment the below line if you see True in the print statement
device = torch.device("cuda:0")
```

```
True
```

First let's load the Stanford NLI dataset from the huggingface datasets hub using the datasets package

Explore the dataset!

```python
from datasets import load_dataset
dataset = load_dataset("snli")
print("Split sizes (num_samples, num_labels):\n", dataset.shape)
print("\nExample:\n", dataset['train'][0])
```

```
Downloading builder script: 100%                    3.82k/3.82k [00:00<00:00, 217kB/s]

Downloading metadata: 100%                          1.90k/1.90k [00:00<00:00, 129kB/s]

Downloading readme: 100%                            14.1k/14.1k [00:00<00:00, 738kB/s]

Downloading: 100%                                   1.93k/1.93k [00:00<00:00, 118kB/s]
```

Each example is a dictionary with the keys: (premise, hypothesis, label).

Data Fields

- premise: a string used to determine the truthfulness of the hypothesis
- hypothesis: a string that may be true, false, or whose truth conditions may not be knowable when compared to the premise
- label: an integer whose value may be either 0, indicating that the hypothesis entails the premise, 1, indicating that the premise and hypothesis neither entail nor contradict each other, or 2, indicating that the hypothesis contradicts the premise.

Create Train, Validation and Test sets

```python
from datasets import load_dataset
from collections import defaultdict

def get_snli(train=10000, validation=1000, test=1000):
    snli = load_dataset('snli')
    train_dataset = get_even_datapoints(snli['train'], train)
    validation_dataset = get_even_datapoints(snli['validation'], validation)
    test_dataset = get_even_datapoints(snli['test'], test)

    return train_dataset, validation_dataset, test_dataset

def get_even_datapoints(datapoints, n):
    random.seed(42)
    dp_by_label = defaultdict(list)
    for dp in tqdm(datapoints, desc='Reading Datapoints'):
        dp_by_label[dp['label']].append(dp)

    unique_labels = [0, 1, 2]

    split = n//len(unique_labels)

    result_datapoints = []

    for label in unique_labels:
        result_datapoints.extend(random.sample(dp_by_label[label], split))

    return result_datapoints

train_dataset, validation_dataset, test_dataset = get_snli()
```

```
Reading Datapoints: 100%                 550152/550152 [00:41<00:00, 18828.97it/s]

Reading Datapoints: 100%                 10000/10000 [00:00<00:00, 17131.92it/s]

Reading Datapoints: 100%                 10000/10000 [00:00<00:00, 17390.18it/s]
```

```python
## sub set stats
from collections import Counter

# num sample stats
print(len(train_dataset), len(validation_dataset), len(test_dataset))

# label distribution
print(Counter([t['label'] for t in train_dataset]))
print(Counter([t['label'] for t in validation_dataset]))
print(Counter([t['label'] for t in test_dataset]))

# We have a perfectly balanced dataset
```

```
9999 999 999
Counter({0: 3333, 1: 3333, 2: 3333})
Counter({0: 333, 1: 333, 2: 333})
Counter({0: 333, 1: 333, 2: 333})
```

We want a function to load samples from the huggingface dataset so that they can be batched and encoded for our model.

Now let's reimplement our tokenizer using the huggingface tokenizer.

Notice that our **call** method (the one called when we call an instance of our class) takes both a premise batch and a hypothesis batch.

The HuggingFace BERT tokenizer knows to join these with the special sentence seperator token between them. We let HuggingFace do most of the work here for making batches of tokenized and encoded sentences.

```
# Nothing to do for this class!

class BatchTokenizer:
    """Tokenizes and pads a batch of input sentences."""

    def __init__(self, model_name='prajjwal1/bert-small'):
        """Initializes the tokenizer

        Args:
            pad_symbol (Optional[str], optional): The symbol for a pad. Defaults to "<P>".
        """
        self.hf_tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.model_name = model_name

    def get_sep_token(self,):
        return self.hf_tokenizer.sep_token

    def __call__(self, prem_batch: List[str], hyp_batch: List[str]) -> List[List[str]]:
        """Uses the huggingface tokenizer to tokenize and pad a batch.

        We return a dictionary of tensors per the huggingface model specification.

        Args:
            batch (List[str]): A List of sentence strings

        Returns:
            Dict: The dictionary of token specifications provided by HuggingFace
        """
        # The HF tokenizer will PAD for us, and additionally combine
        # The two sentences deimited by the [SEP] token.
        enc = self.hf_tokenizer(
            prem_batch,
            hyp_batch,
            padding=True,
            return_token_type_ids=False,
            return_tensors='pt'
        )

        return enc


# HERE IS AN EXAMPLE OF HOW TO USE THE BATCH TOKENIZER
tokenizer = BatchTokenizer()
x = tokenizer(*[["this is the first premise", "This is the second premise"], ["This is first hypothesis", "This is the second hypothesis"]])
print(x)
tokenizer.hf_tokenizer.batch_decode(x["input_ids"])
```

```
(…)wal1/bert-small/resolve/main/config.json: 100%                    286/286 [00:00<00:00, 13.8kB/s]

(…)jjwal1/bert-small/resolve/main/vocab.txt: 100%                    232k/232k [00:00<00:00, 2.52MB/s]
{'input_ids': tensor([[  101,  2023,  2003,  1996,  2034, 18458,   102,  2023,  2003,  2034,
         10744,   102,     0],
        [  101,  2023,  2003,  1996,  2117, 18458,   102,  2023,  2003,  1996,
          2117, 10744,   102]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}
['[CLS] this is the first premise [SEP] this is first hypothesis [SEP] [PAD]',
 '[CLS] this is the second premise [SEP] this is the second hypothesis [SEP]']
```

We can batch the train, validation, and test data, and then run it through the tokenizer

```
def generate_pairwise_input(dataset: List[Dict]) -> (List[str], List[str], List[int]):
    """
    TODO: group all premises and corresponding hypotheses and labels of the datapoints
    a datapoint as seen earlier is a dict of premis, hypothesis and label
    """
    premises = []
    hypothesis = []
    labels = []
    for x in dataset:
        premises.append(x['premise'])
        hypothesis.append(x['hypothesis'])
        labels.append(x['label'])

    return premises, hypothesis, labels


train_premises, train_hypotheses, train_labels = generate_pairwise_input(train_dataset)
validation_premises, validation_hypotheses, validation_labels = generate_pairwise_input(validation_dataset)
test_premises, test_hypotheses, test_labels = generate_pairwise_input(test_dataset)


def chunk(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

def chunk_multi(lst1, lst2, n):
    for i in range(0, len(lst1), n):
        yield lst1[i: i + n], lst2[i: i + n]

batch_size = 16

# Notice that since we use huggingface, we tokenize and
# encode in all at once!
tokenizer = BatchTokenizer()
train_input_batches = [b for b in chunk_multi(train_premises, train_hypotheses, batch_size)]
# Tokenize + encode
train_input_batches = [tokenizer(*batch) for batch in train_input_batches]
```

Let's batch the labels, ensuring we get them in the same order as the inputs

```python
def encode_labels(labels: List[int]) -> torch.FloatTensor:
    """Turns the batch of labels into a tensor

    Args:
        labels (List[int]): List of all labels in the batch

    Returns:
        torch.FloatTensor: Tensor of all labels in the batch
    """
    return torch.LongTensor([int(l) for l in labels])


train_label_batches = [b for b in chunk(train_labels, batch_size)]
train_label_batches = [encode_labels(batch) for batch in train_label_batches]
```

Now we implement the model. Notice the TODO and the optional TODO (read why you may want to do this one.)

```python
class NLIClassifier(torch.nn.Module):
    def __init__(self, output_size: int, hidden_size: int, model_name='prajjwal1/bert-small'):
        super().__init__()
        self.output_size = output_size
        self.hidden_size = hidden_size

        # Initialize BERT, which we use instead of a single embedding layer.
        self.bert = BertModel.from_pretrained(model_name)

        # TODO [OPTIONAL]: Updating all BERT parameters can be slow and memory intensive.
        # Freeze them if training is too slow. Notice that the learning
        # rate should probably be smaller in this case.
        # Uncommenting out the below 2 lines means only our classification layer will be updated.

        for param in self.bert.parameters():
            param.requires_grad = False

        self.bert_hidden_dimension = self.bert.config.hidden_size

        # TODO: Add an extra hidden layer in the classifier, projecting
        #      from the BERT hidden dimension to hidden size. Hint: torch.nn.Linear()

        self.hidden_layer = torch.nn.Linear(self.bert_hidden_dimension, self.hidden_size)

        # TODO: Add a relu nonlinearity to be used in the forward method
        #      https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

        self.relu = torch.nn.ReLU()

        self.classifier = torch.nn.Linear(self.hidden_size, self.output_size)
        self.log_softmax = torch.nn.LogSoftmax(dim=2)

    def encode_text(
        self,
        symbols: Dict
    ) -> torch.Tensor:
        """Encode the (batch of) sequence(s) of token symbols BERT.
            Then, get CLS represenation.

        Args:
            symbols (Dict): The Dict of token specifications provided by the HuggingFace tokenizer

        Returns:
            torch.Tensor: CLS token embedding
        """
        # First we get the contextualized embedding for each input symbol
        # We no longer need an LSTM, since BERT encodes context and
        # gives us a single vector describing the sequence in the form of the [CLS] token.
        encoded_sequence = self.bert(**symbols)
        # TODO: Get the [CLS] token
        #      The BertModel output. See here: https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertModel
        #      and check the returns for the forward method.
        # We want to return a tensor of the form batch_size x 1 x bert_hidden_dimension
        # print(encoded_sequence.last_hidden_state.shape)
        cls_token_embedding = encoded_sequence.last_hidden_state[:, 0, :]
        # Return only the first token's embedding from the last_hidden_state. Hint: using list slices
        return cls_token_embedding.unsqueeze(1)

    def forward(
        self,
        symbols: Dict,
    ) -> torch.Tensor:
        """_summary_

        Args:
            symbols (Dict): The Dict of token specifications provided by the HuggingFace tokenizer

        Returns:
            torch.Tensor: _description_
        """
        encoded_sents = self.encode_text(symbols)
        output = self.hidden_layer(encoded_sents)
        output = self.relu(output)
        output = self.classifier(output)
        return self.log_softmax(output)


# For making predictions at test time
def predict(model: torch.nn.Module, sents: torch.Tensor) -> List:
    logits = model(sents.to(device)).to(device)
    return list(torch.argmax(logits.cpu(), axis=2).squeeze().numpy())
```

Evaluation metrics: Macro F1

```
import numpy as np
from numpy import sum as t_sum
from numpy import logical_and


def precision(predicted_labels, true_labels, which_label=1):
    """
    Precision is True Positives / All Positives Predictions
    """
    pred_which = np.array([pred == which_label for pred in predicted_labels])
    true_which = np.array([lab == which_label for lab in true_labels])
    denominator = t_sum(pred_which)
    if denominator:
        return t_sum(logical_and(pred_which, true_which))/denominator
    else:
        return 0.


def recall(predicted_labels, true_labels, which_label=1):
    """
    Recall is True Positives / All Positive Labels
    """
    pred_which = np.array([pred == which_label for pred in predicted_labels])
    true_which = np.array([lab == which_label for lab in true_labels])
    denominator = t_sum(true_which)
    if denominator:
        return t_sum(logical_and(pred_which, true_which))/denominator
    else:
        return 0.


def f1_score(
    predicted_labels: List[int],
    true_labels: List[int],
    which_label: int
):
    """
    F1 score is the harmonic mean of precision and recall
    """
    P = precision(predicted_labels, true_labels, which_label=which_label)
    R = recall(predicted_labels, true_labels, which_label=which_label)

    if P and R:
        return 2*P*R/(P+R)
    else:
        return 0.


def macro_f1(
    predicted_labels: List[int],
    true_labels: List[int],
    possible_labels: List[int],
    label_map=None
):
    converted_prediction = [label_map[int(x)] for x in predicted_labels] if label_map else predicted_labels
    scores = [f1_score(converted_prediction, true_labels, l) for l in possible_labels]
    # Macro, so we take the uniform avg.
    return sum(scores) / len(scores)
```

Training loop.

```
def training_loop(
    num_epochs,
    train_features,
    train_labels,
    dev_sents,
    dev_labels,
    optimizer,
    model,
):
    print("Training...")
    loss_func = torch.nn.NLLLoss()
    batches = list(zip(train_features, train_labels))
    random.shuffle(batches)
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            preds = model(features.to(device)).squeeze(1)
            loss = loss_func(preds, labels.to(device))
            # Backpropogate the loss through our model
            loss.backward()
            optimizer.step()
            losses.append(loss.item())

        print(f"epoch {i}, loss: {sum(losses)/len(losses)}")
        # Estimate the f1 score for the development set
        print("Evaluating dev...")
        all_preds = []
        all_labels = []
        for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
            pred = predict(model, sents)
            all_preds.extend(pred)
            all_labels.extend(list(labels.cpu().numpy()))

        dev_f1 = macro_f1(all_preds, all_labels, [0, 1, 2])
        print(f"Dev F1 {dev_f1}")

    # Return the trained model
    return model


# You can increase epochs if need be
epochs = 40

# TODO: Find a good learning rate and hidden size
LR = 0.0001
hidden_size = 10

possible_labels = set(train_labels)
model = NLIClassifier(output_size=len(possible_labels), hidden_size=hidden_size)
```

```
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), LR)

batch_tokenizer = BatchTokenizer()

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

training_loop(
    epochs,
    train_input_batches,
    train_label_batches,
    validation_input_batches,
    validation_batch_labels,
    optimizer,
    model,
)
```

```
Training...
100%                                        625/625 [00:07<00:00, 87.28it/s]
epoch 0, loss: 1.097312852668762
Evaluating dev...
100%                                        63/63 [00:00<00:00, 109.09it/s]
Dev F1 0.40836113835216586
100%                                        625/625 [00:06<00:00, 115.30it/s]
epoch 1, loss: 1.0728248240470886
Evaluating dev...
100%                                        63/63 [00:00<00:00, 119.80it/s]
Dev F1 0.43575302162301854
100%                                        625/625 [00:05<00:00, 114.52it/s]
epoch 2, loss: 1.0527944693565368
Evaluating dev...
100%                                        63/63 [00:00<00:00, 127.26it/s]
Dev F1 0.43842869505546433
100%                                        625/625 [00:06<00:00, 84.64it/s]
epoch 3, loss: 1.0374208548545838
Evaluating dev...
100%                                        63/63 [00:00<00:00, 96.55it/s]
Dev F1 0.4481222188173606
100%                                        625/625 [00:05<00:00, 115.46it/s]
epoch 4, loss: 1.0250985927581786
Evaluating dev...
100%                                        63/63 [00:00<00:00, 125.63it/s]
Dev F1 0.46905523666548105
100%                                        625/625 [00:05<00:00, 104.00it/s]
epoch 5, loss: 1.0153348851203918
Evaluating dev...
100%                                        63/63 [00:00<00:00, 103.31it/s]
Dev F1 0.4579154223732537
100%                                        625/625 [00:06<00:00, 121.53it/s]
epoch 6, loss: 1.0078289861679077
Evaluating dev...
100%                                        63/63 [00:00<00:00, 126.80it/s]
Dev F1 0.48763826122372084
100%                                        625/625 [00:05<00:00, 117.71it/s]
epoch 7, loss: 1.000646111011505
Evaluating dev...
100%                                        63/63 [00:00<00:00, 122.60it/s]
Dev F1 0.49600474082283297
100%                                        625/625 [00:05<00:00, 96.46it/s]
epoch 8, loss: 0.9948511468887329
Evaluating dev...
100%                                        63/63 [00:00<00:00, 125.45it/s]
Dev F1 0.5008522962974137
100%                                        625/625 [00:06<00:00, 118.01it/s]
epoch 9, loss: 0.9896235171318054
Evaluating dev...
100%                                        63/63 [00:00<00:00, 127.25it/s]
Dev F1 0.5055315592659991
100%                                        625/625 [00:05<00:00, 122.72it/s]
epoch 10, loss: 0.9849632378578186
Evaluating dev...
100%                                        63/63 [00:00<00:00, 131.09it/s]
Dev F1 0.5089966732987328
100%                                        625/625 [00:06<00:00, 82.73it/s]
epoch 11, loss: 0.9807702584266662
Evaluating dev...
100%                                        63/63 [00:00<00:00, 101.09it/s]
Dev F1 0.5115116317352455
100%                                        625/625 [00:05<00:00, 124.14it/s]
epoch 12, loss: 0.976937613773346
Evaluating dev...
100%                                        63/63 [00:00<00:00, 126.03it/s]
Dev F1 0.5134668202463019
100%                                        625/625 [00:05<00:00, 102.92it/s]
epoch 13, loss: 0.9734506892204284
Evaluating dev...
100%                                        63/63 [00:00<00:00, 107.27it/s]
Dev F1 0.517433780975072
100%                                        625/625 [00:06<00:00, 114.60it/s]
epoch 14, loss: 0.9701675902366638
Evaluating dev...
100%                                        63/63 [00:00<00:00, 122.86it/s]
Dev F1 0.5187565519005393
100%                                        625/625 [00:05<00:00, 114.82it/s]
epoch 15, loss: 0.9670367488861084
Evaluating dev...
100%                                        63/63 [00:00<00:00, 124.02it/s]
Dev F1 0.5205251459625017
100%                                        625/625 [00:06<00:00, 86.04it/s]
epoch 16, loss: 0.9641344951629639
Evaluating dev...
100%                                        63/63 [00:00<00:00, 111.90it/s]
Dev F1 0.5214686795398046
100%                                        625/625 [00:06<00:00, 120.23it/s]
epoch 17, loss: 0.9613689749717712
Evaluating dev...
100%                                        63/63 [00:00<00:00, 121.42it/s]
Dev F1 0.5215130160548425
100%                                        625/625 [00:05<00:00, 112.65it/s]
epoch 18, loss: 0.9587447636604309
```

```
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 126.88it/s]
Dev F1 0.5236197370243185
 100%                                          625/625 [00:06<00:00, 84.59it/s]
epoch 19, loss: 0.9562762427330017
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 92.26it/s]
Dev F1 0.5227544666077187
 100%                                          625/625 [00:05<00:00, 109.63it/s]
epoch 20, loss: 0.9538764311790466
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 124.26it/s]
Dev F1 0.5216824453023359
 100%                                          625/625 [00:05<00:00, 100.09it/s]
epoch 21, loss: 0.9516332839012146
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 110.66it/s]
Dev F1 0.5236656535488264
 100%                                          625/625 [00:06<00:00, 114.43it/s]
epoch 22, loss: 0.9494729369163514
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 124.65it/s]
Dev F1 0.5217824106204634
 100%                                          625/625 [00:06<00:00, 80.58it/s]
epoch 23, loss: 0.94742669506073
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 96.82it/s]
Dev F1 0.524698969843478
 100%                                          625/625 [00:07<00:00, 78.22it/s]
epoch 24, loss: 0.9453948566436767
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 87.10it/s]
Dev F1 0.5256169787277479
 100%                                          625/625 [00:05<00:00, 124.96it/s]
epoch 25, loss: 0.9434526980400085
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 130.63it/s]
Dev F1 0.5277655843934727
 100%                                          625/625 [00:05<00:00, 93.10it/s]
epoch 26, loss: 0.9415931610107422
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 106.13it/s]
Dev F1 0.5266499596912614
 100%                                          625/625 [00:06<00:00, 120.49it/s]
epoch 27, loss: 0.9397716620445251
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 123.42it/s]
Dev F1 0.5255751872214796
 100%                                          625/625 [00:05<00:00, 112.66it/s]
epoch 28, loss: 0.9380079201698304
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 126.27it/s]
Dev F1 0.5246047439854541
 100%                                          625/625 [00:06<00:00, 99.51it/s]
epoch 29, loss: 0.9363240198135376
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 102.41it/s]
Dev F1 0.5249031702620811
 100%                                          625/625 [00:06<00:00, 114.98it/s]
epoch 30, loss: 0.9347326590538025
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 125.40it/s]
Dev F1 0.5268454283782212
 100%                                          625/625 [00:05<00:00, 114.78it/s]
epoch 31, loss: 0.9331026349067688
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 125.49it/s]
Dev F1 0.526944043456965
 100%                                          625/625 [00:06<00:00, 88.98it/s]
epoch 32, loss: 0.9315242610931397
Evaluating dev...
 100%                                          63/63 [00:00<00:00, 120.23it/s]
```

```python
# TODO: Get a final macro F1 on the test set.
# You should be able to mimic what we did with the validaiton set.
batch_tokenizer = BatchTokenizer()

test_input_batches = [b for b in chunk_multi(test_premises, test_hypotheses, batch_size)]

# Tokenize + encode
test_input_batches = [batch_tokenizer(*batch) for batch in test_input_batches]
test_batch_labels = [b for b in chunk(test_labels, batch_size)]
test_batch_labels = [encode_labels(batch) for batch in test_batch_labels]

all_preds = []
all_labels = []
# test_premises, test_hypotheses, test_labels = generate_pairwise_input(test_dataset)
# print(test_premises)
# print(test_input_batches)
# print(test_batch_labels)
for sents, labels in tqdm(zip(test_input_batches, test_batch_labels), total=len(test_input_batches)):
    pred = predict(model, sents)
    all_preds.extend(pred)
    all_labels.extend(list(labels.cpu().numpy()))

dev_f1 = macro_f1(all_preds, all_labels, [0, 1, 2])
print(f"Test F1 {dev_f1}")
```

```
100%                                        63/63 [00:00<00:00, 111.09it/s]
Evaluating dev
```

```
import torch

# Assume 'model' is your PyTorch model
torch.save(model.state_dict(), 'bert-snli.pth')
```

```
    epoch 38, loss: 0.923057128238678
```

Evaluation of Pretrained Model

Huggingface also hosts models which users have already trained on SNLI -- we can load them here and evaluate their performance on the validation and test set.

These models include the BertModel which we were using before, but also the trained weights for the classifier layer. Because of this, we'll use the standard HuggingFace classification model instead of the classifier we used above, and modify the training and prediction functions to handle this correctly.

Try and find the differences between the training loop. One addition is the new usage of "label_map". Why may this be necessary?

```
          (word_embeddings): Embedding(30522, 512, padding_idx=0)
```

```
def class_predict(model, sents):

    with torch.inference_mode():

        logits = model(**sents.to(device)).logits
        predictions = torch.argmax(logits, axis=1)

    return predictions

def prediction_loop(model, dev_sents, dev_labels, label_map=None):
    print("Evaluating...")
    all_preds = []
    all_labels = []
    for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
        pred = class_predict(model, sents).cpu()
        all_preds.extend(pred)
        all_labels.extend(list(labels.cpu().numpy()))

    # print(all_preds)
    # print(all_labels)

    dev_f1 = macro_f1(all_preds, all_labels, possible_labels=set(all_labels), label_map = label_map)
    print(f"F1 {dev_f1}")

def class_training_loop(
    num_epochs,
    train_features,
    train_labels,
    dev_sents,
    dev_labels,
    optimizer,
    model,
    label_map=None
):
    print("Training...")
    loss_func = torch.nn.CrossEntropyLoss()
    batches = list(zip(train_features, train_labels))
    random.shuffle(batches)
    for i in range(num_epochs):
        losses = []
        for features, labels in tqdm(batches):
            # Empty the dynamic computation graph
            optimizer.zero_grad()
            logits = model(**features.to(device)).logits
            # preds = torch.argmax(logits, axis=1)
            loss = loss_func(logits, labels)
            # Backpropogate the loss through our model
            loss.backward()
            optimizer.step()
            losses.append(loss.item())

        print(f"epoch {i}, loss: {sum(losses)/len(losses)}")
        # Estimate the f1 score for the development set
    print("Evaluating dev...")
    all_preds = []
    all_labels = []
    for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
        pred = predict(model, sents).cpu()
        all_preds.extend(pred)
        all_labels.extend(list(labels.numpy()))

    all_preds
    dev_f1 = macro_f1(all_preds, all_labels, possible_labels=set(all_labels), label_map = label_map)
    print(f"Dev F1 {dev_f1}")

    # Return the trained model
    return model
```

Now we can load a model and re-tokenize our data.

```
## TODO: Get the label_map
## For the snli dataset  0=Entailment, 1=Neutral, 2=Contradiction
label_map = {0: 2, 1: 0, 2: 1}
```

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'textattack/bert-base-uncased-snli'
tokenizer_model_name = 'textattack/bert-base-uncased-snli' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map=label_map)
```

```
(…)se-uncased-snli/resolve/main/config.json: 100%          630/630 [00:00<00:00, 17.5kB/s]
pytorch_model.bin: 100%          438M/438M [00:01<00:00, 249MB/s]
(…)base-uncased-snli/resolve/main/vocab.txt: 100%          232k/232k [00:00<00:00, 2.62MB/s]
Evaluating...
100%          63/63 [00:02<00:00, 23.46it/s]
F1 0.6671987159037203
```

To complete this section, find 2 other BERT-based models which have been trained on natural language inference and evaluate them on the dev set. Note the scores for each model and any high-level differences you can find between the models (architecture, sizes, training data, etc.)

If you don't have access to a GPU, inference may be slow, particularly for larger models. In this case, take a sample of the validation set; the size should be large enough such that all labels are covered, and a score will still be meaningful, but also so that inference doesn't take more than 3-5 minutes.

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'boychaboy/SNLI_bert-base-uncased'
tokenizer_model_name = 'boychaboy/SNLI_bert-base-uncased' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map=None)
```

```
(…)rt-base-uncased/resolve/main/config.json: 100%          825/825 [00:00<00:00, 57.6kB/s]
pytorch_model.bin: 100%          438M/438M [00:01<00:00, 260MB/s]
(…)cased/resolve/main/tokenizer_config.json: 100%          285/285 [00:00<00:00, 18.3kB/s]
(…)bert-base-uncased/resolve/main/vocab.txt: 100%          232k/232k [00:00<00:00, 1.33MB/s]
(…)base-uncased/resolve/main/tokenizer.json: 100%          466k/466k [00:00<00:00, 1.84MB/s]
(…)sed/resolve/main/special_tokens_map.json: 100%          112/112 [00:00<00:00, 8.07kB/s]
Evaluating...
100%          63/63 [00:02<00:00, 23.65it/s]
F1 0.6709028992295
```

```
from transformers import  BertForSequenceClassification, BertTokenizer


model_name = 'emrecan/bert-base-multilingual-cased-snli_tr'
tokenizer_model_name = 'emrecan/bert-base-multilingual-cased-snli_tr' # This is sometimes different from model_name, but should normally be the same

model =  BertForSequenceClassification.from_pretrained(model_name)

model.to(device)

batch_tokenizer = BatchTokenizer(model_name = tokenizer_model_name)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]

# Tokenize + encode
validation_input_batches = [batch_tokenizer(*batch) for batch in validation_input_batches]
validation_batch_labels = [b for b in chunk(validation_labels, batch_size)]
validation_batch_labels = [encode_labels(batch) for batch in validation_batch_labels]

prediction_loop(model, validation_input_batches, validation_batch_labels, label_map={0:0, 1:1, 2:2})
```

```
(…)l-cased-snli_tr/resolve/main/config.json: 100%          1.11k/1.11k [00:00<00:00, 59.3kB/s]
pytorch_model.bin: 100%          712M/712M [00:17<00:00, 42.1MB/s]
(…)li_tr/resolve/main/tokenizer_config.json: 100%          341/341 [00:00<00:00, 21.5kB/s]
(…)ual-cased-snli_tr/resolve/main/vocab.txt: 100%          996k/996k [00:00<00:00, 8.40MB/s]
(…)ased-snli_tr/resolve/main/tokenizer.json: 100%          1.96M/1.96M [00:00<00:00, 11.5MB/s]
(…)_tr/resolve/main/special_tokens_map.json: 100%          112/112 [00:00<00:00, 5.43kB/s]
Evaluating...
100%          63/63 [00:03<00:00, 22.02it/s]
F1 0.20654891478929152
```

Written Assignment

1. Describe the task and what capability is required to solve it.

2. How does the method of encoding sequences of words in our model differ here, compared to the word embeddings in HW 4. What is different? Why benefit does this method have?

3. Discuss your results. Did you do any hyperparameter tuning? Did the model solve the task?

1.

The task for this part of the assignment is Natural Language Inference (NLI) also known as Recognizing Textual Entailment (RTE). The goal is to determine the relationship between two given sentences: a premise and a hypothesis. We are looking at three relationships:

- Entailment: The hypothesis logically follows from the fact that the premise is true.
- Contradiction: The hypothesis contradicts the fact that the premise is true.
- Neutral: There is no significant relationship between the premise and the hypothesis.

The technical capabilities needed by the model are:

- Transfer Learning: We are training out model on top of a pretrained BERT model while freezing some layers
- Word Encoding: We are converting the words from strings to a integer vector to represent its meaning
- Fine Tuning: We have to fine tune the hyperparameters to obtain the optimal model

2.

The major difference in encoding between the two parts of assignments is the incorporation of context in the Bert Tokenizer. While we use Word2Vec in the other part that only allows for each word to be represented by a different vector however the word itself only has one vector regardless of the context. Meanwhile the BERT tokenizer takes into account the context to create unique embedding for the word. This leads to the same word having different embedding depending on the context. This method allows for a better understanding of the sense of the word as the context can change the meaning of a word. For example, we could use "beam" in multiple ways, it could be used to refer to a metalic component used in construction such as "Lift the beam carefully, make sure it does not fall", or it could be used to refer to an laser, such that "The Death Star's laser beam obliterated Alderan". The sense of beam is very different in both the sentences which is better encapsulated through the BERT tokeniser.

3.

We have obtained a macro f1 score of 0.53 on the test set. I tuned the hyperparameters to obtain this as the best case. Upon the model having too large a hidden layer or a large learning rate the model seems to get stuck at an f1 score of 0.167 due to only printing 0. However upon fine tuning the model I was able to get past that the model showed marked improvement since then. However it does stop improving near a f1 score of 0.5.

I also looked into other models, while the model provided to us gave an f1 score of 0.667, the model I found by boychaboy got an f1 score of 0.67 while the model by emrecan only gave an f1 score of 0.2. The model given had a different label map than ours and hence I had to pass that, while the model by boychaboy had the same label map. Overall these models perform better that mine, this could be due to them having more resources and time to train the model, while also using the entire bert model instead of freezing some of it like I did due to limited resources.

```
print(model)
```

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(119547, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
```