# 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

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

### Explore the dataset!

In [1]:
```python
# Imports for most of the notebook
import torch
from transformers import BertModel
from transformers import AutoTokenizer
from typing import Dict, List
from util import get_snli
```

In [2]:
```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])
```

```
Found cached dataset snli (/Users/xingyuchen/.cache/huggingface/datasets/snli/plain_text/1.0.0/1f60b67533b65ae0275561ff7828aad5ee4282d0e6f844fd148d05d3c6ea251b)
  0%|          | 0/3 [00:00<?, ?it/s]
Split sizes (num_samples, num_labels):
 {'test': (10000, 3), 'train': (550152, 3), 'validation': (10000, 3)}

Example:
 {'premise': 'A person on a horse jumps over a broken down airplane.', 'hypothesis': 'A person is training his horse for a competition.', 'label': 1}
```

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.

In [3]:
```python
from util import get_snli

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

```
Found cached dataset snli (/Users/xingyuchen/.cache/huggingface/datasets/snli/plain_text/1.0.0/1f60b67533b65ae0275561ff7828aad5ee4282d0e6f844fd148d05d3c6ea251b)
  0%|          | 0/3 [00:00<?, ?it/s]
Reading Datapoints: 100%|██████████| 550152/550152 [00:16<00:00, 32839.24it/s]
Reading Datapoints: 100%|██████████| 10000/10000 [00:00<00:00, 32119.83it/s]
Reading Datapoints: 100%|██████████| 10000/10000 [00:00<00:00, 33953.32it/s]
```

In [4]:
```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.

In [5]:
```python
# Nothing to do for this class!

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

    def __init__(self):
        """Initializes the tokenizer

        Args:
            pad_symbol (Optional[str], optional): The symbol for a pad. Defaults to "<P>".
        """
        self.hf_tokenizer = AutoTokenizer.from_pretrained("prajjwal1/bert-small")

    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 premise.", "This is also a premise"], ["this is the hypothesis", "This is a secon
print(x)
tokenizer.hf_tokenizer.batch_decode(x["input_ids"])
```

```
{'input_ids': tensor([[  101,  2023,  2003,  1996, 18458,  1012,   102,  2023,  2003,  1996,
         10744,   102,     0],
        [  101,  2023,  2003,  2036,  1037, 18458,   102,  2023,  2003,  1037,
          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]])}
2022-11-03 20:53:36.428210: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

Out[5]:
```
['[CLS] this is the premise. [SEP] this is the hypothesis [SEP] [PAD]',
 '[CLS] this is also a premise [SEP] this is a second hypothesis [SEP]']
```

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

In [6]:
```python
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 = []
    label = []
    for row in dataset:
        x, y, z = row.values()
        premises.append(x)
        hypothesis.append(y)
        label.append(z)
    return premises, hypothesis, label
```

In [7]:
```python
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)
```

In [8]:
```python
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]

# Notice that since we use huggingface, we tokenize and
# encode in all at once!
tokenizer = BatchTokenizer()
batch_size = 4
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

In [9]:
```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.)

In [10]:
```python
class NLIClassifier(torch.nn.Module):
    def __init__(self, output_size: int, hidden_size: int):
        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("prajjwal1/bert-small")
        # 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.
        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 with an LSTM.
            Then, get the last (non-padded) hidden state for each symbol and return that.

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

        Returns:
            torch.Tensor: The final hiddens tate of the LSTM, which represents an encoding of
                the entire sentence
        """
        # 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 using the `pooler_output` from
        #      The BertModel output. See here: https://huggingface.co/docs/transformers/model_doc/bert#transfor
        #      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
        last_hidden = encoded_sequence.pooler_output[:, None, :]
        return last_hidden


    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)
```

In [11]:
```python
# For making predictions at test time
def predict(model: torch.nn.Module, sents: torch.Tensor) -> List:
    logits = model(sents)
    return list(torch.argmax(logits, axis=2).squeeze().numpy())
```

### Evaluation metrics: Macro F1

In [12]:
```python
from numpy import logical_and, sum as t_sum
import numpy as np

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]
):
    scores = [f1_score(predicted_labels, true_labels, l) for l in possible_labels]
    # Macro, so we take the uniform avg.
    return sum(scores) / len(scores)
```

### Training loop.

In [13]:
```python
from tqdm import tqdm_notebook as tqdm
import random
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).squeeze(1)
            loss = loss_func(preds, 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)
            all_preds.extend(pred)
            all_labels.extend(list(labels.numpy()))

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

    # Return the trained model
    return model
```

In [20]:
```python
# You can increase epochs if need be
epochs = 10
# TODO: Find a good learning rate
LR = 0.0005

possible_labels = len(set(train_labels))
model = NLIClassifier(output_size=possible_labels, hidden_size = 64)
optimizer = torch.optim.AdamW(model.parameters(), LR)

validation_input_batches = [b for b in chunk_multi(validation_premises, validation_hypotheses, batch_size)]
# Tokenize + encode
validation_input_batches = [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]
```

```
Some weights of the model checkpoint at prajjwal1/bert-small were not used when initializing BertModel: ['cls.predictions.transform.LayerNorm.bias', 'cls.predictions.decoder.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.bias', 'cls.predictions.decoder.bias', 'cls.seq_relationship.weight', 'cls.predictions.transform.dense.weight', 'cls.predictions.bias', 'cls.predictions.transform.dense.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

In [21]:
```python
training_loop(
    epochs,
    train_input_batches,
    train_label_batches,
    validation_input_batches,
    validation_batch_labels,
    optimizer,
    model,
)
```

```
Training...
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_21469/408791926.py:18: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for features, labels in tqdm(batches):
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 0, loss: 1.078682099223137
Evaluating dev...
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_21469/408791926.py:33: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for sents, labels in tqdm(zip(dev_sents, dev_labels), total=len(dev_sents)):
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.3961114104765758
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 1, loss: 1.0384947150826453
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.43722834832583457
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 2, loss: 1.0189619132041932
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.45177991732135503
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 3, loss: 1.0068106312036513
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.4595075784196594
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 4, loss: 0.9973996498465538
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.4715147180501213
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 5, loss: 0.9899630133390427
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.4829240022221357
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 6, loss: 0.9836426736235618
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.48121076978999605
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 7, loss: 0.9784472920656204
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.48489830961978936
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 8, loss: 0.9726479669570923
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.48397059252223357
  0%|          | 0/2500 [00:00<?, ?it/s]
epoch 9, loss: 0.9676756937742234
Evaluating dev...
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.4831066903376536
```

Out[21]:
```
NLIClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 512, padding_idx=0)
      (position_embeddings): Embedding(512, 512)
      (token_type_embeddings): Embedding(2, 512)
      (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=512, out_features=512, bias=True)
              (key): Linear(in_features=512, out_features=512, bias=True)
              (value): Linear(in_features=512, out_features=512, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=512, out_features=512, bias=True)
              (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=512, out_features=2048, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=2048, out_features=512, bias=True)
            (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (1): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=512, out_features=512, bias=True)
              (key): Linear(in_features=512, out_features=512, bias=True)
              (value): Linear(in_features=512, out_features=512, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=512, out_features=512, bias=True)
              (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=512, out_features=2048, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=2048, out_features=512, bias=True)
            (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (2): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=512, out_features=512, bias=True)
              (key): Linear(in_features=512, out_features=512, bias=True)
              (value): Linear(in_features=512, out_features=512, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=512, out_features=512, bias=True)
              (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=512, out_features=2048, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=2048, out_features=512, bias=True)
            (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (3): BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=512, out_features=512, bias=True)
              (key): Linear(in_features=512, out_features=512, bias=True)
              (value): Linear(in_features=512, out_features=512, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=512, out_features=512, bias=True)
              (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=512, out_features=2048, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=2048, out_features=512, bias=True)
            (LayerNorm): LayerNorm((512,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=512, out_features=512, bias=True)
      (activation): Tanh()
    )
  )
  (hidden_layer): Linear(in_features=512, out_features=64, bias=True)
  (relu): ReLU()
  (classifier): Linear(in_features=64, out_features=3, bias=True)
  (log_softmax): LogSoftmax(dim=2)
)
```

In [22]:
```python
# TODO: Get a final macro F1 on the test set.
# You should be able to mimic what we did with the validaiton set.

# test_premises, test_hypotheses, test_labels


test_input_batches = [b for b in chunk_multi(test_premises, test_hypotheses, batch_size)]
# Tokenize + encode
test_input_batches = [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 = []
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.numpy()))

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

```
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_21469/2551025514.py:16: TqdmDeprecationWarning: This function will be removed in tqdm==5.0.0
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  for sents, labels in tqdm(zip(test_input_batches, test_batch_labels), total=len(test_input_batches)):
  0%|          | 0/250 [00:00<?, ?it/s]
Dev F1 0.45765762206965377
```

# Written Assignment

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

Given two sentences: a premise and a hypothesis, classify the relationship between them. Three relationship: Entailment, contradiction, neutral. The dataset has these three values for training. We use HuggingFace BERT tokenizer to tokenize and encode the sentences in batch. For NLI Classifier, we add layers to the model such as ReLU, Linear, LogSoftmax. For encode texrt we use pooler_output from the bertmodel output, which is the last layer hidden-state of the first token of the sequence (classification token) after further processing through the layers used for the auxiliary pretraining task. Then we just hyperparameter tuning for model to get relatively good f1-score.

## 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?

The differences are: 1. we dont need LSTM to encode context and packed the seqence from the LSTM output. The BERT model already encode context and gives us a single vector describing the sequence in the form of the token. 2. We need to preprocess the word embedding in HW4 to get a tensor that indexes the last non-PAD position in each tensor in the batch. However, in BERT we just reshape the dimension to the required output shape. The benefits are we dont need to preprocess the embeddings on our own method and no need to change the dimension of output from model just reshape them. Make the code clean and process quicker.

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

I did hyperparameter tuning for hidden_size and learning rate, the result is showed below, the model solve the task:

epoch = 10
batch size = 8
hidden size = 128

| | loss | dev | test |
|---|---|---|---|
| lr = 0.001 | **0.9846** | **0.5019** | 0.4429 |
| lr = 0.002 | 1.0997 | 0.1666 | 0.1666 |
| lr = 0.005 | 1.0469 | 0.4377 | 0.4612 |
| lr = 0.01 | 1.1041 | 0.1666 | 0.1666 |
| lr = 0.0001 | 0.9863 | 0.4141 | 0.3910 |
| lr = 0.0005 | 0.9856 | 0.4998 | **0.4929** |

epoch = 10
learning rate = 0.0005
hidden size = 128

| | loss | dev | test |
|---|---|---|---|
| batch_size = 8 | 0.9856 | 0.4998 | 0.4929 |
| batch_size = 16 | 1.0051 | 0.4897 | 0.4888 |
| batch_size = 4 | **0.9756** | **0.5024** | **0.4958** |

epoch = 10
learning rate = 0.0005
batch_size = 4

| | loss | dev | test |
|---|---|---|---|
| hidden_size = 256 | **0.9554** | 0.4554 | 0.4408 |
| hidden_size = 64 | 0.9676 | 0.4831 | 0.4576 |
| hidden_size = 128 | 0.9756 | **0.5024** | **0.4958** |

For the best result for now is: epoch = 10
learning rate = 0.0005
batch_size = 4
hidden_size = 128

| loss | dev | test |
|---|---|---|
| 0.9756 | 0.5024 | 0.4958 |

# Prompt Engineering and Probing with GPT3

For the extra-credit, we will be exploring the recent trend that has revolutionalized this field. With GPT3, we can do a variety of tasks without the need of training a model. All we need to do is convert the task into an text generation task that follows a set of instructions called *prompts*. As an example, the task of sentiment classification can be designed as:

```
Decide whether a Tweet's sentiment is positive, neutral, or negativ
e.

Tweet: I loved the new Batman movie!
Sentiment:
```

The GPT3 model then completes the text above with the response **Positive**. The above prompt is an example of zero-shot prediction, meaning, we are not providing any signal/direction that can guide the decision. We could also design the prompt as follows:

```
Decide whether a Tweet's sentiment is positive, neutral, or negativ
e.

Tweet: I really liked the Spiderman movie!
Sentiment: Positive

Tweet: I loved the new Batman movie!
Sentiment:
```

Now this is an example of 1-shot learning, i.e., you are providing an labeled example of how the output should look and then ask GPT3 to complete the next example. When you use more than 1 labeled example, it is known as few-shot learning. The expectation is that, if you provide more examples in the prompt, it will make better predictions.

## Getting Started

In this assignment, we will first need to register for an account at: https://beta.openai.com/ (https://beta.openai.com/) As a free trial, you will get $18 credits to make api calls to the GPT3 server. Once registered, you should go through the docs here: https://beta.openai.com/docs/guides/completion/prompt-design (https://beta.openai.com/docs/guides/completion/prompt-design) to get more info on the capabilities of the model. You can then go directly interact with GPT3 in the playground: https://beta.openai.com/playground (https://beta.openai.com/playground). For making these calls programmatically, we will do the following:

In [12]:
```
# pip install openai
```

In [13]:
```
import os

## Find the API key by clicking on your profile in the openai page. Add the
## Make sure to delete this cell afterwords

os.environ['OPENAI_API_KEY'] = ''
```

In [14]:
```
import os
import openai

openai.api_key = os.getenv('OPENAI_API_KEY')

response = openai.Completion.create(
  model="text-davinci-002",
  prompt="Decide whether a Tweet's sentiment is positive, neutral, or negat
  temperature=0,
  max_tokens=60,
  top_p=1,
  frequency_penalty=0.5,
  presence_penalty=0
)
```

In [15]:
```
response
```

Out[15]:
```
<OpenAIObject text_completion id=cmpl-66PhJU2oLwwnFlf0lA3j9L3dBapTJ at 0x
7f7b0ee38630> JSON: {
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": " Positive"
    }
  ],
  "created": 1666986769,
  "id": "cmpl-66PhJU2oLwwnFlf0lA3j9L3dBapTJ",
  "model": "text-davinci-002",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 1,
    "prompt_tokens": 31,
    "total_tokens": 32
  }
}
```

In [16]:
```
response['choices'][0]['text']
```

Out[16]:
```
' Positive'
```

If you see ' Positive' as response in the above cell, you have successfully set-up gpt3 in your system.

Now, the task for the assignment is really just do something cool. For example, you could probe how well GPT3 performs on the tasks in the previous HWs. Or, you could do something like question-answering or summarization, that were not covered in the assignments. The choice is yours.

# Submission

Please submit a written report of what task you tried probing, how well did GPT3 do for that task and what were your key takeaways in this experiment.

In HW5, Task A we given two sentences: a premise and a hypothesis, classify the relationship between them. Three relationship: Entailment, contradiction, neutral.

There are three sections in the dataset:

Split sizes (num_samples, num_labels): {'test': (10000, 3), 'train': (550152, 3), 'validation': (10000, 3)}

Example: {'premise': 'A person on a horse jumps over a broken down airplane.', 'hypothesis': 'A person is training his horse for a competition.', 'label': 1}

In hw5, Task B I want to use GPT3 to go further about hw5 task A. In the shared task, our team choose Task 4 human value argument, which is kind like hw5 task A. The shared task given three sentences: a premise, a hypothesis, and a relationship, we need to classify the human value behind these three sentences.

Thre are totally 20 labels (human values):

1. Self-direction: thought
2. Self-direction: action
3. Stimulation
4. Hedonism
5. Achievement
6. Power: dominance
7. Power: resources
8. Face
9. Security: personal
10. Security: societal
11. Tradition
12. Conformity: rules
13. Conformity: interpersonal
14. Humility
15. Benevolence: caring
16. Benevolence: dependability
17. Universalism: concern
18. Universalism: nature
19. Universalism: tolerance

20. Universalism: objectivity

Feature Dataset
ArgumentID Conclusion Stance Premise
A01010 We should prohibit school prayer against it should be allowed if the student wants to pray as long as it is not interfering with his classes
A01011 We should abolish the three-strikes laws in favor of three strike laws can cause young people to be put away for life without a chance to straight out their life
A01012 The use of public defenders should be mandatory in favor of the use of public defenders should be mandatory because some people don't have money for a lawyer and this would help those that don't

Target Dataset
Argument ID Self-direction: thought Self-direction: action Stimulation Hedonism Achievement Power: dominance Power: resources Face Security: personal Security: societal Tradition Conformity: rules Conformity: interpersonal Humility Benevolence: caring Benevolence: dependability Universalism: concern Universalism: nature Universalism: tolerance Universalism: objectivity
A01010 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
A01011 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 1
A01012 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

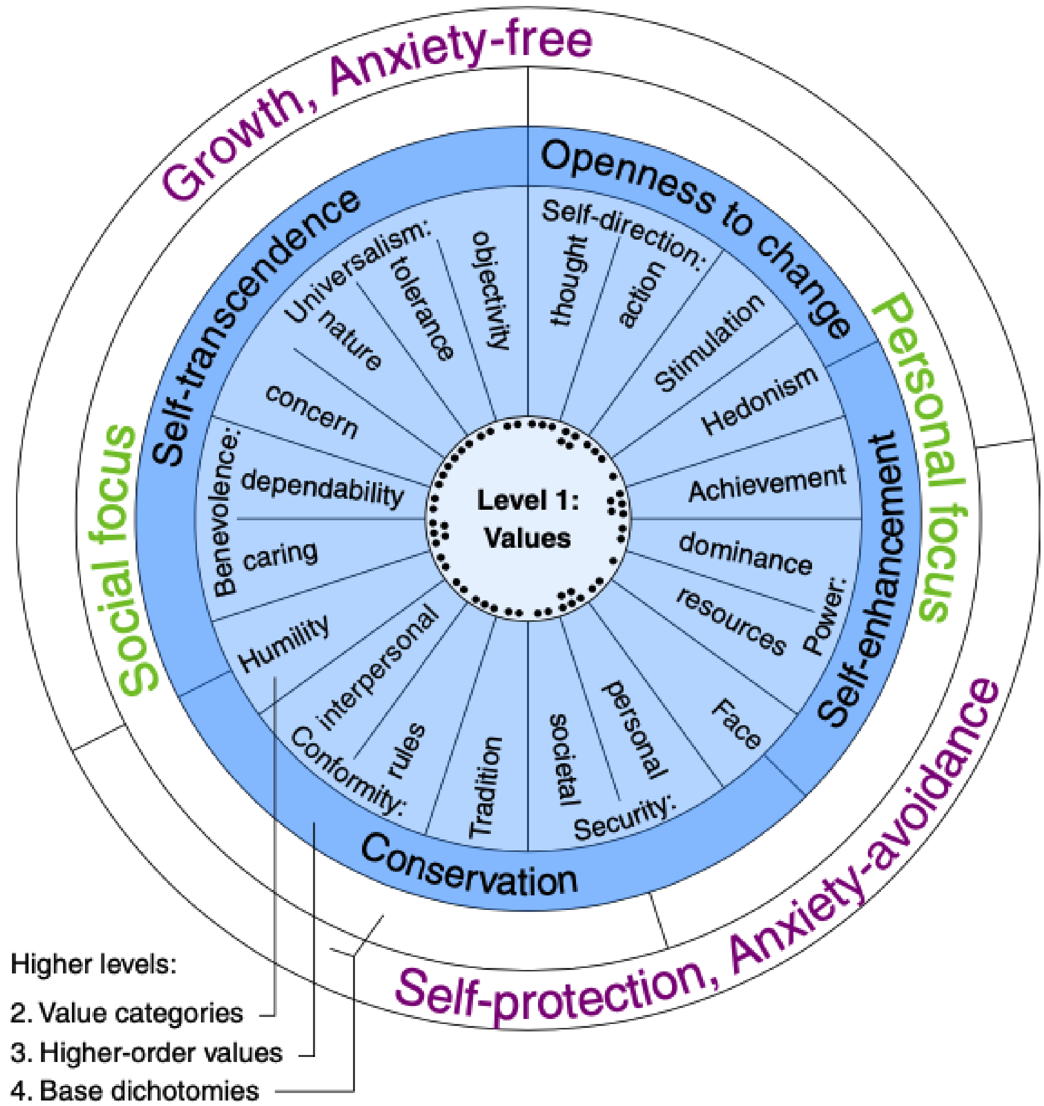
However, the shared task has a big drawback is that it has relatively small dataset, it has only 5220 samples.

So, I decide to enrich the dataset by using GPT3 to manually label human value on the snli dataset from hw5 task A.

I will combine snli dataset (test, train, validation) into 570152 samples, and try to use GPT3 to label one of human values to them.

Here is the example for doing that, I will not process 570152 sample because it will overuse free $18 credits for openai api. So just check if GPT3 can handle this task.

In order to easily interpret the result, I use the higher level label described in the paper to minimize the 20 lables into 4 labels: self-transcendence, openness to change, self-enhancement, conservation.




In [3]:
```
import pandas as pd

labels_df = pd.read_csv('labels-training.tsv', sep='\t')
```

In [4]:
```
# Self-transcendence
labels_df.loc[(labels_df['Universalism: concern'] == 1)
                                    | (labels_df['Universalism: nature'] ==
                                   | (labels_df['Universalism: tolerance'] =
                                   | (labels_df['Universalism: objectivity']
                                   | (labels_df['Benevolence: caring'] == 1)
                                   | (labels_df['Benevolence: dependability'
```

Out[4]:

| | Argument ID | Self-direction: thought | Self-direction: action | Stimulation | Hedonism | Achievement | Power: dominance | Power: resources | Fa |
|---|---|---|---|---|---|---|---|---|---|
| **6** | A01007 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **7** | A01008 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **8** | A01009 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **9** | A01010 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | |
| **10** | A01011 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |

5 rows × 21 columns

In [5]:
```
# Conservation
labels_df.loc[(labels_df['Conformity: interpersonal'] == 1)
                                    | (labels_df['Conformity: rules'] == 1)
                                   | (labels_df['Tradition'] == 1)
                                   | (labels_df['Security: personal'] == 1)
                                   | (labels_df['Security: societal'] == 1)]
```

Out[5]:

| | Argument ID | Self-direction: thought | Self-direction: action | Stimulation | Hedonism | Achievement | Power: dominance | Power: resources | Fac |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A01001 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **1** | A01002 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **3** | A01004 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **4** | A01005 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **5** | A01006 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |

5 rows × 21 columns

In [6]:
```
# Self-enhancement
labels_df.loc[(labels_df['Achievement'] == 1)
                                        | (labels_df['Power: dominance'] == 1)
                                       | (labels_df['Power: resources'] == 1)].h
```

Out[6]:

| | Argument ID | Self-direction: thought | Self-direction: action | Stimulation | Hedonism | Achievement | Power: dominance | Power: resources | Fa |
|---|---|---|---|---|---|---|---|---|---|
| **5** | A01006 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| **10** | A01011 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |
| **16** | A01017 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |
| **17** | A01018 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | |
| **52** | A03013 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | |

5 rows × 21 columns

In [8]:
```
# Openness to change
labels_df.loc[(labels_df['Self-direction: thought'] == 1)
                                        | (labels_df['Self-direction: action'] =
                                       | (labels_df['Stimulation'] == 1)].head()
```

Out[8]:

| | Argument ID | Self-direction: thought | Self-direction: action | Stimulation | Hedonism | Achievement | Power: dominance | Power: resources | Fa |
|---|---|---|---|---|---|---|---|---|---|
| **2** | A01003 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| **9** | A01010 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | |
| **19** | A01020 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |
| **21** | A02002 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |
| **31** | A02012 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | |

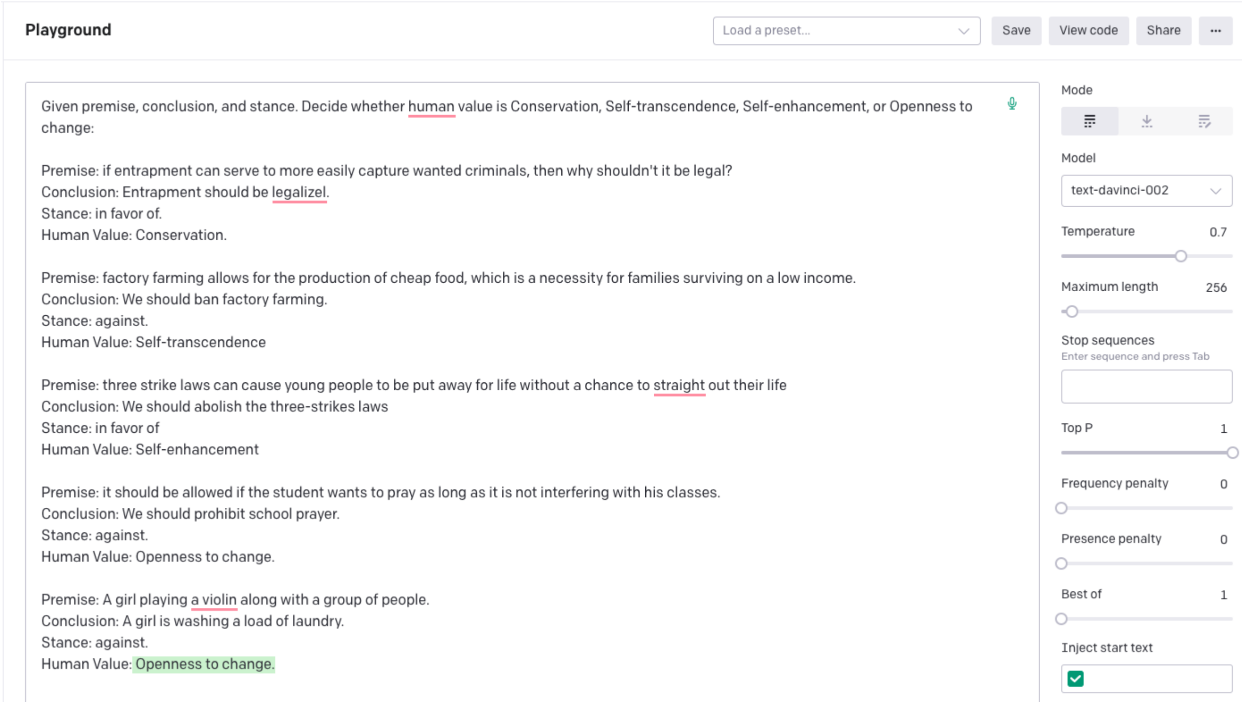
5 rows × 21 columns

## zero-shot prediction

Given premise, conclusion, and stance. Decide whether human value is Conservation, Self-transcendence, Self-enhancement, or Openness to change:

Premise: A girl playing a violin along with a group of people.
Conclusion: A girl is washing a load of laundry.
Stance: against.
Human Value: Self-transcendence

## 1-shot learning




few-shot learning

Premise: if entrapment can serve to more easily capture wanted criminals, then why shouldn't it be legal?
Conclusion: Entrapment should be legalizel.
Stance: in favor of.
Human Value: Conservation.

Premise: nuclear weapons help keep the peace in uncertain times.
Conclusion: We should fight for the abolition of nuclear weapons.
Stance: against.
Human Value: Conservation.

Premise: factory farming allows for the production of cheap food, which is a necessity for families surviving on a low income.
Conclusion: We should ban factory farming.
Stance: against.
Human Value: Self-transcendence.

Premise: we should ban human cloning as it will only cause huge issues when you have a bunch of the same humans running around all acting the same.
Conclusion: We should ban human cloning.
Stance: in favor of.
Human Value: Self-transcendence.

Premise: three strike laws can cause young people to be put away for life without a chance to straight out their life.
Conclusion: We should abolish the three-strikes laws.
Stance: in favor of.
Human Value: Self-enhancement.

Premise: affirmative action is no longer necessary as more minorities are able to prove to employers that they are worthy workers.
Conclusion: We should end affirmative action.
Stance: in favor of.
Human Value: Self-enhancement.

Premise: it should be allowed if the student wants to pray as long as it is not interfering with his classes.
Conclusion: We should prohibit school prayer.
Stance: against.
Human Value: Openness to change.

Premise: It is important for news organizations to transfer to new forms of media, like the internet, but it is costly and requires subsidization.
Conclusion: We should subsidize journalism.
Stance: in favor of.
Human Value: Openness to change.

Premise: A girl playing a violin along with a group of people.
Conclusion: A girl is washing a load of laundry.
Stance: against.
Human Value: Openness to change.

It seems that 1 shot and few shot approach with GPT3 gives relatively feasible result but not for zero shot.