INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY HYDERABAD

ADVERSARIAL NLI

ASHNA DUA (2021101072)
ASHNA.DUA@STUDENTS.IIIT.AC.IN
VANSHITA MAHAJAN (2021101102)
VANSHITA.MAHAJAN@STUDENTS.IIIT.AC.IN
PRISHA (2021101075)
PRISHA.KUMAR@RESEARCH.IIIT.AC.IN

OCTOBER 7, 2024

Contents

1

Problem Statement

1.1 Introduction

Large-scale pre-trained language models like BERT and RoBERTa have significantly advanced performance in numerous Natural Language Processing (NLP) tasks, establishing themselves as essential tools in the field. Despite their remarkable capabilities, these models exhibit vulnerabilities when exposed to adversarial textual inputs.

This creates a critical challenge, as even small perturbations in the input can lead to dramatic changes in model output, compromising their reliability in sensitive applications. Traditional fine-tuning approaches have fallen short in mitigating these weaknesses, leaving room for new, more resilient strategies.

In this project, we tackle the problem from an information-theoretic perspective. Our goal is to develop a robust fine-tuning framework that not only preserves the models' high performance but also increases their resistance to adversarial examples. By introducing mutual-information-based regularizers, we aim to reduce noise in the feature representations while reinforcing critical features, thus ensuring the model can maintain accuracy under both standard and adversarial training conditions.

1.2 Dataset: Adversarial Natural Language Inference (ANLI)

The Adversarial Natural Language Inference (ANLI) dataset is a large-scale, challenging benchmark designed for natural language understanding (NLU) tasks, specifically natural language inference (NLI). ANLI was collected using a unique iterative, adversarial human-and-model-in-the-loop process aimed at creating examples that expose model weaknesses, ensuring continuous learning and robustness.

1.2.1 Collection Process

The dataset was collected in three rounds (A1, A2, and A3), with increasing levels of difficulty in each round. The data collection followed the **Human-And-Model-in-the-Loop Enabled Training (HAMLET)** process, which involves human annotators

attempting to create hypothesis examples (given a context and a target label) that state-of-the-art models misclassify. The process includes:

- **Context Selection:** Annotators were provided with short, multi-sentence passages (context) from sources such as Wikipedia, news articles, and other genres.
- **Hypothesis Generation:** Annotators wrote hypotheses corresponding to a target label (*entailment, contradiction, or neutral*) that aimed to fool the model into incorrect classifications.
- **Verification:** Hypotheses that fooled the model were then verified by human annotators to ensure that the label was correct and the example was valid.

The dataset includes both verified correct examples and those where the model failed, to continuously challenge the evolving model.

1.2.2 Rounds of Data Collection

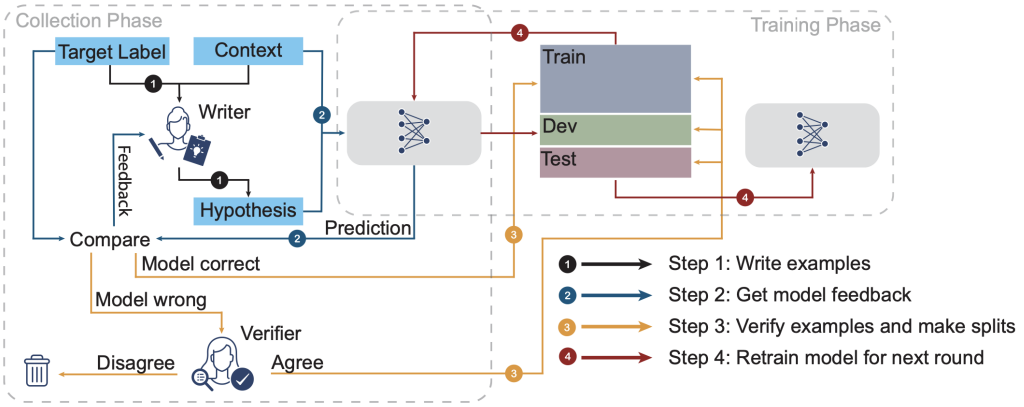

Figure 1: Adversarial NLI data collection via human-and-model-in-the-loop enabled training (HAMLET). The four steps make up one round of data collection. In step 3, model-correct examples are included in the training set; development and test sets are constructed solely from model-wrong verified-correct examples.

Figure 1.1: *Rounds for Data Collection*

Each round in ANLI represents a progressively more difficult phase of model training and evaluation:

- **A1 (Round 1):** The first round used contexts sampled from Wikipedia and the HotpotQA dataset. A BERT-Large model was employed as the baseline model for this round.
- **A2 (Round 2):** In the second round, a stronger RoBERTa model was used, incorporating the training data from Round 1 along with new data.
- **A3 (Round 3):** The third round involved a more diverse set of contexts from domains such as news articles, fiction, spoken text from court transcripts, and procedural texts, making it the most challenging.

The increasing difficulty in each round stems from the iterative nature of the data collection process, where models are strengthened and tested on progressively harder examples generated by human annotators.

To prevent overfitting to specific annotator styles or model characteristics, the test set for each round includes an **exclusive subset**. This exclusive subset consists of examples from annotators who did not contribute to the training data, ensuring that models are tested on unseen data from entirely new annotators. This prevents models from learning biases inherent to specific annotators and ensures better generalization across varied inputs.

1.2.3 Dataset Statistics

ANLI contains a total of 162,865 training examples collected over three rounds, with 3,200 examples in both the development and test sets for each round. The dataset is balanced to ensure no overfitting occurs due to model bias or specific genre characteristics. Table 1.1 provides a detailed breakdown of the dataset's statistics.

Round	Train Size	Dev Size	Test Size
A1	16,946	1,000	1,000
A2	45,460	1,000	1,000
A3	100,459	1,200	1,200

Table 1.1: *Dataset statistics for ANLI.*

1.2.4 Comparison to Other Datasets

ANLI represents an improvement over prior NLI datasets like SNLI and MNLI by focusing on more complex and longer contexts. While SNLI used image captions as contexts, ANLI sources multi-sentence passages from various genres, adding greater complexity. The human-and-model-in-the-loop process also makes ANLI more robust against overfitting, allowing models trained on this dataset to better handle real-world NLI tasks by addressing more nuanced and challenging examples.

2

Literature Review

Textual adversarial attacks have emerged as a significant concern for language models, particularly those based on pre-trained transformers like BERT and RoBERTa. The existing adversarial attacks predominantly focus on word-level manipulations. Ebrahimi et al., 2018 were among the first to introduce a white-box, gradient-based approach to search for adversarial word or character substitutions. Subsequent research (Alzantot et al., 2018; Ren et al., 2019; Zang et al., 2020; Jin et al., 2020) sought to refine these methods by restricting the perturbation search space and using Part-of-Speech (POS) checking to ensure that the adversarial examples maintain a natural appearance to human observers.

To defend against such adversarial attacks, three primary defense strategies have been proposed:

- **Adversarial Training**: A popular and practical defense mechanism, adversarial training augments the model with adversarial examples. Some works (C. Zhu et al., 2020; Jiang et al., 2020; Liu et al., 2020; Gan et al., 2020) employ PGD-based (Projected Gradient Descent) attacks in the embedding space to generate adversarial examples for data augmentation or use virtual adversarial training to regularize the objective function. However, the primary limitation of this approach is its dependency on the threat model, which makes it less effective against unseen attacks.
- **Interval Bound Propagation (IBP)**: Introduced by Dvijotham et al., 2018, IBP is designed to handle worst-case perturbations theoretically. Recent research (Huang et al., 2019; Jia et al., 2017) has extended this approach to the NLP domain, applying IBP to certify the robustness of language models. However, IBP relies on strong assumptions about model architecture, making it difficult to apply to modern transformer-based models.
- **Randomized Smoothing**: Cohen et al., 2019 introduced randomized smoothing as a method to guarantee robustness in l_2 norm by adding Gaussian noise to smooth the classifier. Ye et al., 2020 adapted this idea to the NLP domain by replacing Gaussian noise with synonym words, ensuring that adversarial word substitutions

within predefined synonym sets maintain robustness. However, ensuring the completeness of the synonym set presents a challenge in practice.

S. Zhu et al., 2020 addressed this issue by proposing a robust representation learning approach that considers the mutual-information perspective in the context of worst-case perturbation. However, their work primarily focused on continuous input spaces like those in computer vision. In contrast, InfoBERT adopts a novel information-theoretic perspective that is designed for both standard and adversarial training in the discrete input space typical of language models.

The existing literature highlights the progress made in defending against textual adversarial attacks and improving the robustness of language models through various techniques. However, most prior work has focused on either adversarial training or theoretical robustness certifications, both of which come with limitations in terms of model adaptability or threat model assumptions. InfoBERT takes a step forward by integrating mutual-information-based regularizers into the fine-tuning process, which not only enhances robustness in adversarial settings but also aligns with standard training requirements for discrete NLP input spaces. This combination of information theory and adversarial training positions InfoBERT as a unique contribution to the ongoing effort to make language models more robust and reliable.

3

Interim Report

3.1 Introduction

The goal of this project is to explore the robustness of pre-trained language models when applied to adversarially generated data in the domain of Natural Language Inference (NLI). We aim to enhance this robustness by incorporating mutual-information-based regularization techniques through the **InfoBERT** training objective. However, before introducing these advanced methods, we first establish a strong baseline by implementing and evaluating a widely used model, the **BERT-Uncased-Base** model.

3.2 Dataset Overview

The dataset used for this project is the **Adversarial Natural Language Inference (ANLI)** dataset, which is specifically designed to test the resilience of NLI models against adversarial examples. ANLI is composed of three rounds, with each subsequent round containing examples that are progressively more difficult for models to handle:

- **ANLI Round 1 (r1):** Contains adversarial examples that exploit basic weaknesses in NLI models.
- **ANLI Round 2 (r2):** More complex than r1, r2 includes examples targeting the models trained in the previous round.
- **ANLI Round 3 (r3):** The most difficult round, r3 has examples designed to challenge the deep reasoning abilities of NLI models.

In this analysis, the labels used in the dataset are defined as follows:

- 0: entailment
- 1: neutral
- 2: contradiction

3.3 Exploratory Data Analysis

	uid	premise	hypothesis	label	reason
0	0fd0abfb-659e-4453-b196-c3a64d2d8267	The Parma trolleybus system (Italian: "Rete fi...	The trolleybus system has over 2 urban routes	0	NaN
1	7ed72ff4-40b7-4f8a-b1b9-6c612aa62c84	Alexandra Lendon Bastedo (9 March 1946 – 12 Ja...	Sharron Macready was a popular character throu...	1	NaN
2	5d2930a3-62ac-485d-94d7-4e36cbbcd7b5	Alexandra Lendon Bastedo (9 March 1946 – 12 Ja...	Bastedo didn't keep any pets because of her vi...	1	NaN
3	324db753-ddc9-4a85-a825-f09e2e5aebdd	Alexandra Lendon Bastedo (9 March 1946 – 12 Ja...	Alexandra Bastedo was named by her mother.	1	NaN
4	4874f429-da0e-406a-90c7-22240ff3ddf8	Alexandra Lendon Bastedo (9 March 1946 – 12 Ja...	Bastedo cared for all the animals that inhabit...	1	NaN

Figure 3.1: *Sample Train Data*

	uid	premise	hypothesis	label	reason
0	4aae63a8-fcf7-406c-a2f3-50c31c5934a9	Ernest Jones is a British jeweller and watchma...	The first Ernest Jones store was opened on the...	0	The first store was opened in London, which is...
1	c577b92c-78fb-4e1d-ae1d-34133609c142	Old Trafford is a football stadium in Old Traf...	There are only 10 larger football stadiums in ...	0	The text says that it is the 11th largest foot...
2	26936cd9-1a5a-4a2b-9fca-899d61880ca0	Magnus is a Belgian joint dance project of Tom...	"The body gave you everything" album was not r...	0	it was released on March 29, 2004. "not this b...
3	cd977941-273b-4748-a5d2-6c7234a2a302	Shadowboxer is a 2005 crime thriller film dire...	Shadowboxer was written and directed by Lee Da...	1	It is not know who wrote the Shadowboxer. The ...
4	1a9eae8f-27d9-47ba-80b8-7d1402ee524a	Takaaki Kajita (梶田 隆章 , Kajita Takaaki) is a ...	Arthur B. McDonald is a Japanese physicist, kn...	2	Arthur B. McDonald is Canadian in the context.

Figure 3.2: *Sample Test Data*

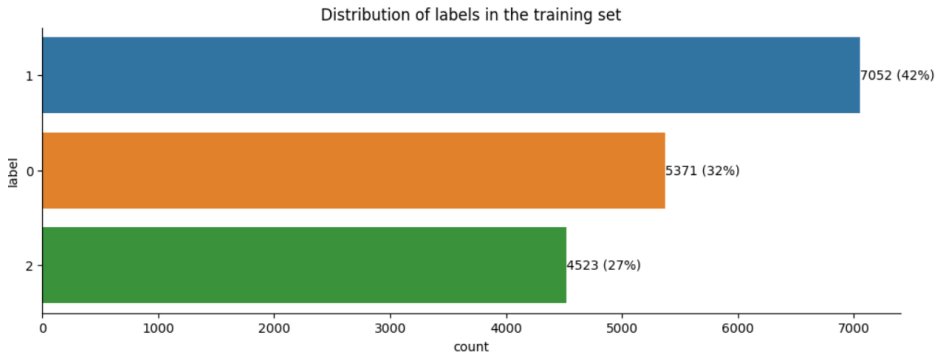

Figure 3.3: *Label Distribution*

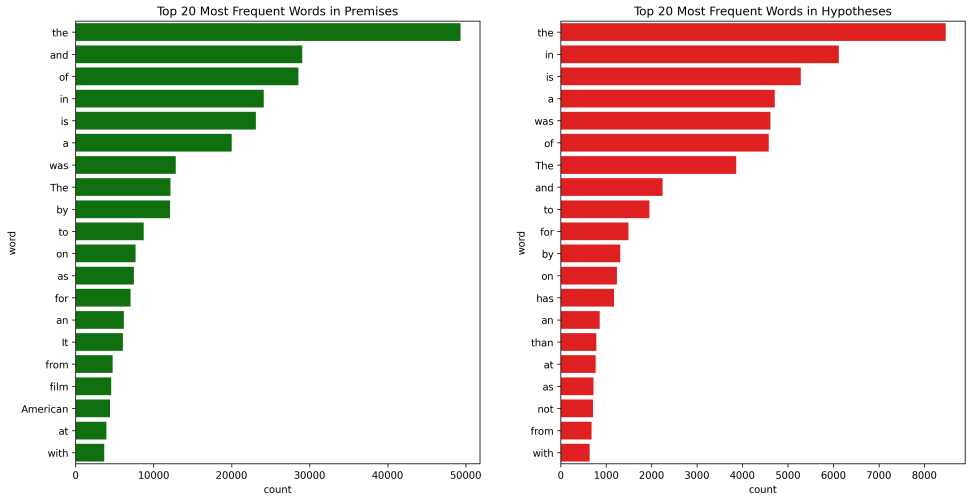

Figure 3.4: *Top 20 most frequent words in Train and Test*

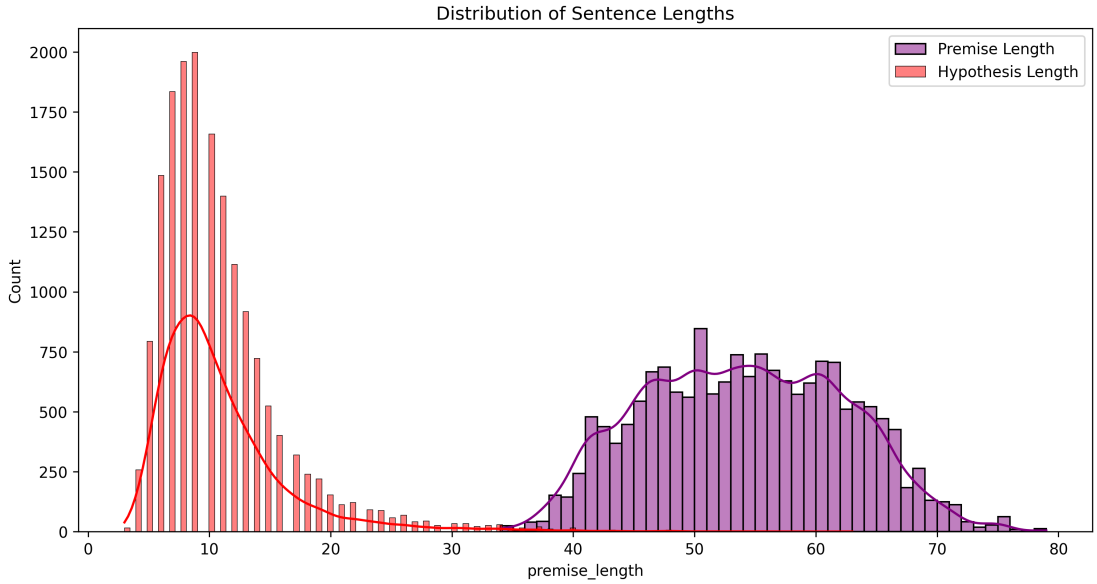

Figure 3.5: *Sentence length distribution in premise and hypothesis*

3.4 Baseline Model: BERT-Uncased-Base

As a starting point, we implemented the **BERT-Uncased-Base** model as the baseline for our experiments. This model has 12 layers, 768 hidden dimensions, and 12 attention heads. It is pre-trained on a large corpus of text and fine-tuned on the ANLI dataset to perform natural language inference.

3.4.1 Training Setup

We fine-tuned the BERT-Uncased-Base model on the ANLI dataset for 30,000 iterations. During this process, the model learned to classify the relationship between pairs of sentences as either *entailment*, *contradiction*, or *neutral*. The training was conducted across all three rounds of the ANLI dataset, and the accuracy on the validation sets was recorded at various checkpoints to assess the model's performance across rounds.

3.5 Training Process

The provided code implements a training pipeline for fine-tuning a pre-trained language model on the Adversarial Natural Language Inference (ANLI) dataset. The code is structured into several components:

3.5.1 Code

```
MODEL_CLASSES = {
    "bert-base": {
        "model_name": "bert-base-uncased",
        "tokenizer": BertTokenizer,
        "sequence_classification": BertForSequenceClassification,
        # "padding_token_value": 0,
        "padding_segement_value": 0,
        "padding_att_value": 0,
        "do_lower_case": True,
    }}
registered_path = {
    'anli_r1_train': config.PRO_ROOT / "data/build/anli/r1/train.jsonl",
    'anli_r1_dev': config.PRO_ROOT / "data/build/anli/r1/dev.jsonl",
    'anli_r1_test': config.PRO_ROOT / "data/build/anli/r1/test.jsonl",

    'anli_r2_train': config.PRO_ROOT / "data/build/anli/r2/train.jsonl",
    'anli_r2_dev': config.PRO_ROOT / "data/build/anli/r2/dev.jsonl",
    'anli_r2_test': config.PRO_ROOT / "data/build/anli/r2/test.jsonl",

    'anli_r3_train': config.PRO_ROOT / "data/build/anli/r3/train.jsonl",
    'anli_r3_dev': config.PRO_ROOT / "data/build/anli/r3/dev.jsonl",
    'anli_r3_test': config.PRO_ROOT / "data/build/anli/r3/test.jsonl",
}

nli_label2index = {
    'e': 0,
    'n': 1,
    'c': 2,
    'h': -1,
}

class NLIDataset(Dataset):
    def __init__(self, data_list, transform) -> None:
        super().__init__()
        self.d_list = data_list
        self.len = len(self.d_list)
        self.transform = transform

    def __getitem__(self, index: int):
        return self.transform(self.d_list[index])

    def __len__(self) -> int:
        return self.len
```

```
class NLITransform(object):
    def __init__(self, model_name, tokenizer, max_length=None):
        self.model_name = model_name
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __call__(self, sample):
        processed_sample = dict()
        processed_sample['uid'] = sample['uid']
        processed_sample['gold_label'] = sample['label']
        processed_sample['y'] = nli_label2index[sample['label']]

        # premise: str = sample['premise']
        premise: str = sample['context'] if 'context' in sample else
        ↪  sample['premise']
        hypothesis: str = sample['hypothesis']

        if premise.strip() == '':
            premise = 'empty'

        if hypothesis.strip() == '':
            hypothesis = 'empty'

        tokenized_input_seq_pair = self.tokenizer.encode_plus(premise,
        ↪  hypothesis,
                                                              max_length=self.ma⌋
                                                              ↪  x_length,
                                                              return_token_type_⌋
                                                              ↪  ids=True,
                                                              ↪  truncation=Tru⌋
                                                              ↪  e)

        processed_sample.update(tokenized_input_seq_pair)

        return processed_sample
```

```
for epoch in tqdm(range(num_epoch), desc="Epoch", disable=args.global_rank not in
↪  [-1, 0]):
        training_list = []
        for i in range(len(train_data_list)):
            print("Build Training Data ...")
            train_d_list = train_data_list[i]
            train_d_name = train_data_name[i]
            train_d_weight = train_data_weights[i]
            cur_train_list = sample_data_list(train_d_list, train_d_weight)
            print(f"Data Name:{train_d_name}; Weight: {train_d_weight}; "
                  f"Original Size: {len(train_d_list)}; Sampled Size:
                  ↪  {len(cur_train_list)}")
            training_list.extend(cur_train_list)

        random.shuffle(training_list)
        train_dataset = NLIDataset(training_list, data_transformer)

        train_sampler = SequentialSampler(train_dataset)
        if not args.cpu and not args.single_gpu:
            print("Use distributed sampler.")
            train_sampler = DistributedSampler(train_dataset, args.world_size,
            ↪  args.global_rank,
                                               shuffle=True)

        train_dataloader = DataLoader(dataset=train_dataset,
                                      batch_size=batch_size_per_gpu_train,
                                      shuffle=False,  #
                                      num_workers=0,
                                      pin_memory=True,
                                      sampler=train_sampler,
                                      collate_fn=BaseBatchBuilder(batching_schem
                                      ↪  a))

        print(debug_node_info(args), "epoch: ", epoch)

        if not args.cpu and not args.single_gpu:
            if args.sampler_seed == -1:
                train_sampler.set_epoch(epoch)
            else:
                train_sampler.set_epoch(epoch + args.sampler_seed)

        for forward_step, batch in enumerate(tqdm(train_dataloader,
        ↪  desc="Iteration",
                                                  disable=args.global_rank not in
                                                  ↪  [-1, 0]), 0):
            model.train()

            batch = move_to_device(batch, local_rank)
            if args.model_class_name in ["distilbert", "bart-large"]:
                outputs = model(batch['input_ids'],
                                attention_mask=batch['attention_mask'],
                                labels=batch['y'])
```

```
            else:
                outputs = model(batch['input_ids'],
                                attention_mask=batch['attention_mask'],
                                token_type_ids=batch['token_type_ids'],
                                labels=batch['y'])
            loss, logits = outputs[:2]

            if args.gradient_accumulation_steps > 1:
                loss = loss / args.gradient_accumulation_steps

            if args.fp16:
                with amp.scale_loss(loss, optimizer) as scaled_loss:
                    scaled_loss.backward()
            else:
                loss.backward()

            if (forward_step + 1) % args.gradient_accumulation_steps == 0:
                if args.max_grad_norm > 0:
                    if args.fp16:
                        torch.nn.utils.clip_grad_norm_(amp.master_params(optimiz⌋
                        ↪  er),
                        ↪  args.max_grad_norm)
                    else:
                        torch.nn.utils.clip_grad_norm_(model.parameters(),
                        ↪  args.max_grad_norm)

                optimizer.step()
                scheduler.step()  # Update learning rate schedule
                model.zero_grad()

                global_step += 1

                if args.global_rank in [-1, 0] and args.eval_frequency > 0 and
                ↪  global_step % args.eval_frequency == 0:
                    r_dict = dict()

                    # Eval Loop
                    for i in range(len(eval_data_name)):
                        cur_eval_data_name = eval_data_name[i]
                        cur_eval_data_list = eval_data_list[i]
                        cur_eval_dataloader = eval_data_loaders[i]
                        # cur_eval_raw_data_list = eval_raw_data_list[i]

                        evaluation_dataset(args, cur_eval_dataloader,
                        ↪  cur_eval_data_list, model, r_dict,
                                           eval_name=cur_eval_data_name)
```

```
                # saving checkpoints
                current_checkpoint_filename = \
                    f'e({epoch})|i({global_step})'

                for i in range(len(eval_data_name)):
                    cur_eval_data_name = eval_data_name[i]
                    current_checkpoint_filename += \
                        f'|{cur_eval_data_name}#({round(r_dict[cur_eval_data_
                        ↪  _name]["acc"],
                        ↪  4)})'

                if not args.debug_mode:
                    # save model:
                    model_output_dir = checkpoints_path /
                    ↪  current_checkpoint_filename
                    if not model_output_dir.exists():
                        model_output_dir.mkdir()
                    model_to_save = (
                        model.module if hasattr(model, "module") else model
                    )

                    torch.save(model_to_save.state_dict(),
                    ↪  str(model_output_dir / "model.pt"))
                    torch.save(optimizer.state_dict(), str(model_output_dir /
                    ↪  "optimizer.pt"))
                    torch.save(scheduler.state_dict(), str(model_output_dir /
                    ↪  "scheduler.pt"))

                # save prediction:
                if not args.debug_mode and args.save_prediction:
                    cur_results_path = prediction_path /
                    ↪  current_checkpoint_filename
                    if not cur_results_path.exists():
                        cur_results_path.mkdir(parents=True)
                    for key, item in r_dict.items():
                        common.save_jsonl(item['predictions'],
                        ↪  cur_results_path / f"{key}.jsonl")

                    # avoid saving too many things
                    for key, item in r_dict.items():
                        del r_dict[key]['predictions']
                    common.save_json(r_dict, cur_results_path /
                    ↪  "results_dict.json", indent=2)
            if args.total_step > 0 and global_step == t_total:
                # if we set total step and global step s t_total.
                is_finished = True
                break
```

```
        if args.global_rank in [-1, 0] and args.total_step <= 0:
            r_dict = dict()
            # Eval loop:
            for i in range(len(eval_data_name)):
                cur_eval_data_name = eval_data_name[i]
                cur_eval_data_list = eval_data_list[i]
                cur_eval_dataloader = eval_data_loaders[i]

                evaluation_dataset(args, cur_eval_dataloader, cur_eval_data_list,
                ↪  model, r_dict,
                                   eval_name=cur_eval_data_name)

            # saving checkpoints
            current_checkpoint_filename = \
                f'e({epoch})|i({global_step})'

            for i in range(len(eval_data_name)):
                cur_eval_data_name = eval_data_name[i]
                current_checkpoint_filename += \
                    f'|{cur_eval_data_name}#({round(r_dict[cur_eval_data_name]["
                    ↪  acc"],
                    ↪  4)})'

            if not args.debug_mode:
                # save model:
                model_output_dir = checkpoints_path / current_checkpoint_filename
                if not model_output_dir.exists():
                    model_output_dir.mkdir()
                model_to_save = (
                    model.module if hasattr(model, "module") else model
                )

                torch.save(model_to_save.state_dict(), str(model_output_dir /
                ↪  "model.pt"))
                torch.save(optimizer.state_dict(), str(model_output_dir /
                ↪  "optimizer.pt"))
                torch.save(scheduler.state_dict(), str(model_output_dir /
                ↪  "scheduler.pt"))

            if not args.debug_mode and args.save_prediction:
                cur_results_path = prediction_path / current_checkpoint_filename
                if not cur_results_path.exists():
                    cur_results_path.mkdir(parents=True)
                for key, item in r_dict.items():
                    common.save_jsonl(item['predictions'], cur_results_path /
                    ↪  f"{key}.jsonl")

                # avoid saving too many things
                for key, item in r_dict.items():
                    del r_dict[key]['predictions']
                common.save_json(r_dict, cur_results_path / "results_dict.json",
                ↪  indent=2)

        if is_finished:
            break
```

3.5.2 Initialization and Data Handling

Model Classes and Data Paths

The script begins by defining key dictionaries:

- **MODEL_CLASSES:** A dictionary that stores configuration details for various pre-trained language models supported by the Hugging Face Transformers library. Each entry in the dictionary corresponds to a specific model (e.g., BERT, XLNet, RoBERTa) and contains information like:
  - **model_name:** The name of the pre-trained model (e.g., "bert-base-uncased").
  - **tokenizer:** The class for the corresponding tokenizer from Transformers.
  - **sequence_classification:** The class for sequence classification using the chosen model.
  - **Other parameters:** Additional parameters specific to the model, such as padding values or whether to lowercase input text.
- **registered_path:** A dictionary that maps dataset names to their corresponding file paths. This provides a convenient way to access data files by using their names.
- **nli_label2index:** A dictionary that maps natural language inference (NLI) labels (e, n, c for entailment, neutral, and contradiction) to their corresponding numerical indices (0, 1, 2).

Data Loading and Preprocessing

The code utilizes custom classes and functions for handling and preparing the ANLI dataset:

- **set_seed:** A function that sets random seeds for reproducibility across different training runs.
- **NLIDataset:** A custom dataset class that wraps the ANLI data, providing a convenient interface for loading and accessing data samples.
- **NLITransform:** A data transformation class that preprocesses each data sample, performing the following steps:
  - Extracts relevant information, such as the sample's unique identifier (uid), the ground truth label (gold_label), and the input text (premise and hypothesis).
  - Uses the chosen tokenizer from Transformers to tokenize the text, converting it into a sequence of numerical IDs.
  - Applies padding to ensure all sequences have the same length and truncation if necessary.
  - Returns a processed dictionary containing the preprocessed text representations, labels, and other relevant data.
- **build_eval_dataset_loader_and_sampler:** Creates a PyTorch DataLoader and Sampler for evaluation purposes, taking a list of data samples, a data transformation function, a batching schema, and the evaluation batch size as input.

- **sample_data_list:** A function that implements a sampling strategy to balance different training data sources. This function can be customized to incorporate different sampling strategies like oversampling minority classes or weighted sampling.

3.5.3 Training Process

Main Function and Command-Line Arguments

The main function serves as the entry point for the script. It parses command-line arguments that control the training process:

- **cpu:** If set, training is performed on the CPU.
- **single_gpu:** If set, training is performed on a single GPU.
- **fp16:** If set, training uses mixed precision (fp16) for potential performance gains.
- **experiment_name:** Specifies the name of the experiment for logging and saving results.
- **resume_path:** Specifies a path to a checkpoint file for resuming training from a previous state.
- **epochs:** Sets the number of training epochs.
- **total_step:** Sets the total number of steps for training.
- **per_gpu_train_batch_size:** Sets the batch size per GPU during training.
- **gradient_accumulation_steps:** Accumulates gradients over multiple batches before updating model parameters.
- **per_gpu_eval_batch_size:** Sets the batch size per GPU during evaluation.
- **max_length:** Sets the maximum length of input sequences to be processed by the model.
- **warmup_steps:** Specifies the number of steps during which the learning rate is gradually increased.
- **max_grad_norm:** Sets the maximum gradient norm for gradient clipping.
- **learning_rate:** Sets the initial learning rate for the optimizer.
- **weight_decay:** Sets the weight decay parameter for the optimizer.
- **adam_epsilon:** Sets the epsilon parameter for the AdamW optimizer.
- **eval_frequency:** Sets the frequency at which the model is evaluated on validation sets.
- **train_data:** Specifies the paths to the training data files.
- **train_weights:** Specifies the weights for different training data sources (if applicable).
- **eval_data:** Specifies the paths to the evaluation data files.

Train Function: The Core of the Training Loop

The train function implements the core training loop:

1. Initialization: Initializes the model, optimizer, scheduler, and data loaders based on the provided arguments.
2. Training Loop:
   (a) Iterates through epochs.
   (b) For each epoch:
      i. Loads and shuffles the training data, creating a DataLoader.
      ii. Iterates through batches of training data.
      iii. Performs a forward pass, calculating the loss.
      iv. Performs a backward pass, computing gradients.
      v. Applies gradient clipping (if enabled).
      vi. Updates model parameters using the optimizer.
      vii. Adjusts the learning rate using the scheduler.
      viii. Evaluation: At regular intervals (defined by eval_frequency), evaluates the model on validation sets and saves predictions and results.
3. Checkpoint Saving: Periodically saves model weights, optimizer state, and scheduler state for resuming training later.

Key Training Parameters and Their Impact

- **Learning Rate:** A critical hyperparameter affecting convergence. An optimal learning rate can lead to faster training and improved model performance.
- **Batch Size:** The number of training samples processed before the model's parameters are updated. A larger batch size can lead to more stable gradients but may require more memory.
- **Warmup Steps:** A period at the beginning of training during which the learning rate is gradually increased. Proper warmup can help the model stabilize before training at a higher learning rate.

3.6 Results

The BERT-Uncased-Base model achieved an accuracy as shown in the table below on the validation set after training on the ANLI dataset for 30,000 iterations. This performance will serve as a benchmark against which we will evaluate the improvements achieved with the InfoBERT training objective.

Round	Accuracy
R1	0.555 (55.5%)
R2	0.452 (45.2%)
R3	0.4492 (44.92%)

3.7 Next Phase

In the next phases of the project, we will implement the InfoBERT training objective and evaluate its effectiveness in enhancing the robustness of the model against adversarial examples. We will also explore various mutual-information-based regularization techniques to further improve the model's resilience.

InfoBERT introduces two mutual-information-based regularizers to enhance the robustness of pre-trained language models:

- **Information Bottleneck (IB) Regularizer:** This regularizer reduces the noisy mutual information between the input and feature representations, removing irrelevant details that may introduce adversarial vulnerability.
- **Anchored Feature Regularizer:** This regularizer enhances the mutual information between local stable features and global features, ensuring that useful and stable features are prioritized over noisy or adversarially modified ones.

The InfoBERT framework will be implemented and fine-tuned on the ANLI dataset to assess its performance against adversarially generated examples.

3.8 Conclusion

This interim report outlines the progress made in the project, focusing on the implementation of a baseline model for NLI using the ANLI dataset. The training of the BERT-Uncased-Base model has been completed, setting the stage for future enhancements aimed at improving model robustness.

The code for this baseline was implemented using Facebook's ANLI git repo, where the terminal arguments were tweaked to train the model.

4 Timeline

Here is the proposed timeline of our project.

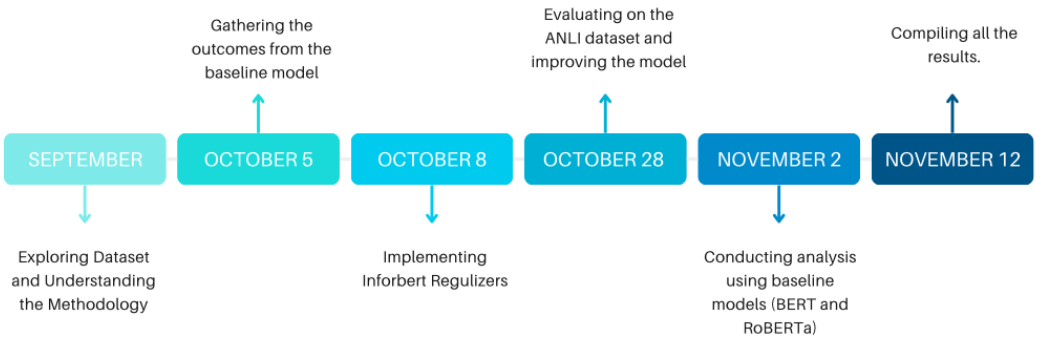

Figure 4.1: *Timeline for the project*

Bibliography

Alzantot, Moustafa et al. (Oct. 2018). "Generating Natural Language Adversarial Examples". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Ed. by Ellen Riloff et al. Brussels, Belgium: Association for Computational Linguistics, pp. 2890–2896. DOI: 10.18653/v1/D18-1316. URL: https://aclanthology.org/D18-1316.

Cohen, Jeremy M, Elan Rosenfeld, and J. Zico Kolter (2019). *Certified Adversarial Robustness via Randomized Smoothing*. arXiv: 1902.02918 [cs.LG]. URL: https://arxiv.org/abs/1902.02918.

Dvijotham, Krishnamurthy et al. (2018). *Training verified learners with learned verifiers*. arXiv: 1805.10265 [cs.LG]. URL: https://arxiv.org/abs/1805.10265.

Ebrahimi, Javid et al. (July 2018). "HotFlip: White-Box Adversarial Examples for Text Classification". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Iryna Gurevych and Yusuke Miyao. Melbourne, Australia: Association for Computational Linguistics, pp. 31–36. DOI: 10.18653/v1/P18-2006. URL: https://aclanthology.org/P18-2006.

Gan, Zhe et al. (2020). *Large-Scale Adversarial Training for Vision-and-Language Representation Learning*. arXiv: 2006.06195 [cs.CV]. URL: https://arxiv.org/abs/2006.06195.

Huang, Po-Sen et al. (Nov. 2019). "Achieving Verified Robustness to Symbol Substitutions via Interval Bound Propagation". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Ed. by Kentaro Inui et al. Hong Kong, China: Association for Computational Linguistics, pp. 4083–4093. DOI: 10.18653/v1/D19-1419. URL: https://aclanthology.org/D19-1419.

Jia, Robin and Percy Liang (Sept. 2017). "Adversarial Examples for Evaluating Reading Comprehension Systems". In: *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Ed. by Martha Palmer, Rebecca Hwa, and Sebastian Riedel. Copenhagen, Denmark: Association for Computational Linguistics, pp. 2021–2031. DOI: 10.18653/v1/D17-1215. URL: https://aclanthology.org/D17-1215.

Jiang, Haoming et al. (July 2020). "SMART: Robust and Efficient Fine-Tuning for Pre-trained Natural Language Models through Principled Regularized Optimization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, pp. 2177–2190. DOI: 10.18653/v1/2020.acl-main.197. URL: https://aclanthology.org/2020.acl-main.197.

Jin, Di et al. (2020). *Is BERT Really Robust? A Strong Baseline for Natural Language Attack on Text Classification and Entailment*. arXiv: 1907.11932 [cs.CL]. URL: https://arxiv.org/abs/1907.11932.

Liu, Xiaodong et al. (2020). *Adversarial Training for Large Neural Language Models*. arXiv: 2004.08994 [cs.CL]. URL: https://arxiv.org/abs/2004.08994.

Ren, Shuhuai et al. (July 2019). "Generating Natural Language Adversarial Examples through Probability Weighted Word Saliency". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, pp. 1085–1097. DOI: 10.18653/v1/P19-1103. URL: https://aclanthology.org/P19-1103.

Ye, Mao, Chengyue Gong, and Qiang Liu (July 2020). "SAFER: A Structure-free Approach for Certified Robustness to Adversarial Word Substitutions". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, pp. 3465–3475. DOI: 10.18653/v1/2020.acl-main.317. URL: https://aclanthology.org/2020.acl-main.317.

Zang, Yuan et al. (July 2020). "Word-level Textual Adversarial Attacking as Combinatorial Optimization". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Online: Association for Computational Linguistics, pp. 6066–6080. DOI: 10.18653/v1/2020.acl-main.540. URL: https://aclanthology.org/2020.acl-main.540.

Zhu, Chen et al. (2020). *FreeLB: Enhanced Adversarial Training for Natural Language Understanding*. arXiv: 1909.11764 [cs.CL]. URL: https://arxiv.org/abs/1909.11764.

Zhu, Sicheng, Xiao Zhang, and David Evans (2020). *Learning Adversarially Robust Representations via Worst-Case Mutual Information Maximization*. arXiv: 2002.11798 [cs.LG]. URL: https://arxiv.org/abs/2002.11798.