Prototypical Pre-Training for Robust Multi-Task Learning in Natural Language Processing

Stanford CS224N {Default} Project

Rohan Sikand
Department of Computer Science
Stanford University
rsikand@stanford.edu

Andre Yeung
Department of Bioengineering
Stanford University
ayyeung@stanford.edu

Abstract

Multi-task learning (MTL) is an efficient approach for jointly training models to perform multiple related tasks all at once. BERT is a general purpose transformer model which is capable of producing embeddings for sequence inputs, as well as serving as a backbone for natural language processing tasks. In this paper, we propose a novel pre-training method to produce robust sentence embeddings for downstream multi-task fine-tuning. Our proposed method, coined prototypical pre-training, harnesses the concept of prototypical networks (Snell et al., 2017) to learn a feature space. We use BERT to act as an embedding function to map input vectors into this feature space. We then fine-tune the overall multi-task model in a jointly optimized fashion and compare our pre-training method to several other methods including a supervised, no pre-trained baseline. Our proposed method was the most performant, achieving an average score of 0.724 on the dev set, and improved upon the supervised baseline which achieved an average score of 0.552 across the three tasks. We argue that prototypical pre-training introduces a new paradigm to produce robust sentence embeddings and offers a new avenue for pre-training which can be further built upon in the future.

1 Key Information to include

- Mentor: Xiaoyuan Ni
- External Collaborators (if you have any): n/a
- Sharing project: n/a

2 Introduction

In this project, we aim to construct and implement a general purpose, BERT transformer model that is able to learn robust sentence embeddings which are useful for improving performance across downstream tasks of different domains including sentiment analysis, paraphrase prediction, and semantic textual similarity analysis. This work helps to better characterize language models, which have gained significant traction in recent times, and shed light on their inner mechanisms to potentially improve them. To this end, we explore multiple methods and the subspace of their hybridization in an attempt to improve their performance beyond our baseline results. This traditionally supervised multi-task learning baseline is constructed from the pre-trained and featurized BERT embeddings, layered downstream with one multi-task perceptron head for each task. While this performs moderately well on sentiment analysis and boolean paraphrase prediction (over 50%), it lacks in the semantic similarity task and generally has room for improvement across all three tasks.

We conjecture that, by combining different methods such as optimizing for cosine similarity and a new pre-training method that we introduce, we can improve these results of the fine-tuned language

model in downstream multi-task learning. Multi-task learning (MTL) is an efficient approach for jointly training models to perform multiple related tasks all at once. In this paper, we propose a method for multi-task learning to learn a feature space with BERT using a method termed prototypical pre-training. Specifically, our proposed method harnesses the concept of prototypical networks Snell et al. (2017) which have the ability to learn task-specific embeddings and prototype representations. We train the BERT backbone to act as an embedding function. We use this embedding function to compute prototypes for each class over the course of training. A new data point is classified by measuring the euclidean distance between each class prototype and the latent vector representation obtained by the embedding function. We conduct pre-training using this method a in supervised fashion which gives us initial parameters for the BERT embedding function for which we harness in downstream fine-tuning.

We evaluate our proposed method on NLP tasks including sentiment analysis, sentence similarity, and paraphrase detection in a multi-task fashion. We then compare the results of our proposed approach to all other methods, including our baseline which is traditional supervised multi-task learning.

3 Related Work

Prototypical networks The main paper for which we build off of is (Snell et al., 2017) which first introduces the general concept of prototypical networks. Specifically, (Snell et al., 2017) propose a neural network architecture called Prototypical Networks (ProtoNets) for few-shot learning, where the goal is to learn from very few examples of a new class. The proposed architecture learns a metric space where distances between samples can be used to classify new instances. It represents each class by the mean of the embedding of its samples in the metric space, and during training, it learns to minimize the distance between the embedding of each sample and its class mean. During inference, the network is given a few labeled examples of a new class, and it computes the class means for each example. It then predicts the label of a test instance as the label of the closest class mean in the embedding space, which the authors demonstrated to exceed the performance of other state-of-the-art few-shot algorithms.

We take inspiration from this paper to formulate our novel method. Specifically, we adapt prototypical learning for pre-training instead of for few-shot learning. That is, we harness the general metric-based idea for learning a robust feature embedding function instead of for performing few-shot classification. To the best of our knowledge, no method exists in literature that uses prototypical learning for pre-training to produce robust sentence embeddings for multi-task learning.

ProtoTransformers While it seems that no previous literature exists regarding using prototypical learning for pre-training explicitly, there are several papers that harness the concept of prototypical networks for few-shot learning in natural language processing tasks. One such paper is (Wu et al., 2021), which introduces the concept of ProtoTransformers for the few-shot learning task of providing feedback to students in natural language. While this paper involves using prototypical networks and BERT, it uses these concepts for few-shot learning on natural language rather than for explicitly pre-training to produce a sentence embedder as we do in this project.

4 Approach

The goal of this project is to construct and implement a multi-task model that harnesses BERT to learn sentence embeddings and model parameters that are robust across several downstream tasks of different domains. To this end, we propose several methods, including a novel one (defined below), and compare this method to a supervised baseline. Thus, the goal of this project is to improve upon the standard supervised baseline across the tasks of paraphrase detection (PARA), semantic textual similarity (STS), sentiment analysis (SST).

4.1 Baseline

To serve as a baseline, we implement traditional multi-task learning for the tasks of paraphrase detection, semantic textual similarity, sentiment analysis. Our baseline is supervised fine-tuning of the multi-task model that uses BERT as a backbone. Specifically, we define f as BERT which serves as a feature extractor to produce sentence embeddings. We then define several multi-layer perceptron

heads for each of the three tasks (one MLP for each of PARA, STS, SST). Let p be the MLP head for PARA, s be the MLP head for STS, and t be the MLP head for SST. Thus, each forward pass is $m(f(x))$ where m is the respective MLP head and x is the input. For the baseline, we freeze the parameters of f and jointly optimize the parameters of p, s, and t using the training data in a supervised fashion. Specifically, for each batch, we compute the loss for each of the three tasks and sum them up to make one loss for which we optimize throughout the course of training. The architecture details of each MLP head are enumerated in appendix A.1.1.

4.2 Modifications to the baseline

In addition to the above method, we tested methods with slight modifications to the supervised baseline. In this section, we describe these altered methods.

Unfrozen fine-tuning. In this approach, we perform the same method as described in the baseline, but do not freeze the parameters of the pretrained BERT backbone f. That is, we let the parameters of f be updated during the course of training.

Cosine Similarity for STS. In this method, we forego the use of a linear head t for the STS task. Instead, we pass the embeddings through a cosine similarity function which calculates the cosine similarity between sentence a and sentence b. From here, we push the unnormalized output through a rectified linear unit (ReLU) to enforce the output to be between 0 and 1 instead of between -1 and 1. Then, we scale the values by 5 such that the final output is between 0 to 5. Thus, the ReLU and scaling combination enforces the final output to be between 0 and 5 which matches the range of the labels for the STS task. We also note that we keep the parameters of f unfrozen in this method.

Weighted Multitask Loss. In this method, instead of summing the three losses together on each update, we perform a weighted summation of the three. Specifically, we define the weighted summation loss, L, as

$$L = \lambda_1 l_p + \lambda_2 l_s + \lambda_3 l_t$$

where l_s is the loss for SST, l_p is the loss for PARA, and l_t is the loss for STS. In our experiments, we tried three weight combinations: (**SST weighted**: $\lambda_1 = 0.98$, $\lambda_1 = 0.01$, $\lambda_1 = 0.01$), (**PARA weighted**: $\lambda_1 = 0.01$, $\lambda_1 = 0.98$, $\lambda_1 = 0.01$), and (**STS weighted**: $\lambda_1 = 0.01$, $\lambda_1 = 0.01$, $\lambda_1 = 0.98$). We also keep the parameters of f unfrozen in this method.

Sharing Layers. Since the MLP heads for STS and PARA have hidden linear layers of the same dimensions, we experimented with *sharing* these layers between the two MLP heads. That is, the same parameters are used for the hidden layers in STS and PARA. Theoretically, we are learning a common representation space between the two tasks for the linear hidden layers. We keep the parameters of f unfrozen in this method.

Concatenation Before Encoding In the previous methods, for the sentence pair tasks where there is two inputs (PARA and STS), we first encode the inputs using the BERT backbone f. Before the last linear layer in the MLP heads, we concatenate using multiplication. However, we hypothesize that encoding the two inputs separately via BERT results in no cross-attention between the two inputs which can result in lower performance. To test this hypothesis, in this approach, we concatenate the two inputs, separated by a `[SEP]` token, before passing the inputs into the BERT encoder f. Thus, cross-attention is now being performed between the two inputs in the BERT encoder.

4.3 Proposed approach: prototypical pre-training

We harness the concept of prototypical networks (protonets) Snell et al. (2017) and adapt this metric learning technique for the multi-task setting in natural language processing tasks. We note that the raw form of protonets only works for classification problems. As such, we only use protonets on the tasks of PARA and SST. For STS, we attach a trainable MLP head as defined in the baseline. At a high-level, the main difference between the baseline (traditional supervised multi-task learning) and this new approach is that we compute the logits differently. From here, cross entropy is applied for

the loss and everything else in the training remains the same. In this section, we will introduce the general concept of prototypes and then we will describe the method for computing the logits for each task, mathematically, below.

Prototypical networks Snell et al. (2017) offer a technique to produce a learned feature space that is divided into regions based on classes. Specifically, the idea behind prototypical networks is to learn embeddings and prototype representations, which capture the similarities and differences between the different classes explicitly. Specifically, prototypical networks learn a prototype for each class by computing the mean of the embeddings of the training examples belonging to that class. During inference, the input is embedded using the learned feature extractor and assigned to the closest prototype, which serves as the prediction for the corresponding task. We now describe this mathematically.

Let f_ϕ be an embedding function (feature extractor), with learnable parameters ϕ, which maps input vectors $\mathbf{x_i}$ to latent vectors in $\mathbf{R}^d$ where d is the hidden dimension of the embedding function. The goal of prototypical networks is to learn optimal parameters ϕ such that inputs of the same class, when transformed by f_ϕ, are close together in this latent space. In this setting, we use BERT as f_ϕ, the embedding function. For each task, we compute a **prototype**, c_k for each class over the course of training on a per-batch basis. This is computed by averaging all of the embedded inputs (after being transformed by f_ϕ) for each class (in the batch) in latent space. Specifically, each protoype is calculates as follows:

$$c_k = \frac{1}{|S_k|} \sum_{x_i \in S_k} f_\phi \left(x_i \right).$$

this produces a d-dimensional representation for each class. S_k is defined as all of the vectors belonging to class k in the current batch. From here, to produce the logits for some input vector x_i, we first calculate the latent vector q as follows:

$$q = f_\phi \left(x_i \right).$$

Then, we take the negative euclidean distance between q and each prototype c_k defined as follows:

$$- \left\| q - c_k \right\|_2 .$$

Combining these values into a set gives us the unnormalized logits. We can then apply softmax over this set to produce a proper probability distribution which gives us the prediction probabilities for each class:

$$P \left(y_i = k \mid x_i \right) = \frac{\exp \left(- \left\| q - c_k \right\|_2 \right)}{\sum_{k'} \exp \left(- \left\| q - c_{k'} \right\|_2 \right)}$$

For PARA and SST, we harness this concept of prototypical networks to compute the predictions for each new input point after training. However, since protonets are not able to perform non-classification based tasks, we modify the STS task by rounding the continious values to discrete values. In this sense, we modified the STS task to be a classification task such that we can use prototypical pre-training on this task as well. From here, we jointly optimize the entire model by summing together the three losses across the tasks and backpropogating through this loss via the AdamW optimizer (Loshchilov and Hutter, 2017). This phase represents the prototypical pre-training phase of the pipeline which gives us an initial set of parameters for f, the BERT backbone. We then fine-tune the linear heads p, t and keep f's parameters unfrozen. Also, we chose to concatenate the inputs before passing the inputs into the BERT encoder f for STS and PARA for both the prototypical pre-training and the fine-tuning phases.

In sum, using metric-based prototypical networks gives us a way to compute the classification of specific examples explicitly (via euclidean distance) instead of via fully-connected neural network layers. We hypothesize that this constraint offers a way to learn a sentence-level feature space where semantically similar sentences are close together, measured by euclidean distance. By enforcing similar sentences to appear close together in this learned space, we are effectively learning "sentence

embeddings" using BERT. By performing this method, we can effectively "pre-train" the parameters of our BERT backbone f in a supervised manner such that downstream fine-tuning of the entire model performs more robustly. We argue that this is a fundamentally different and novel way of pre-training as, traditionally, BERT is trained in a self-supervised manner using techniques like masked language modeling.

5 Experiments

5.1 Data

We harness the three datasets given for each task. For SST, we use the Stanford Sentiment Treebank (SST) dataset. For PARA, we use the Quora dataset. For STS, we use SemEval STS dataset. We follow the default configured train/dev/test splits.

For the SST task, the model is given an input sentence and classifies the discrete sentiment of the sentence as either negative (0), somewhat negative (1), neutral (2), somewhat positive (3), or positive (4).

For the PARA task, the model is given a pair of sentences and must classify whether the sentences are a paraphrases of each other. Thus, this is a binary classification task given two inputs.

For the STS task, the model is given a pair of sentences and predicts how similar the two sentences are on a continious scale from 0 to 5.

5.2 Evaluation method

We evaluate the approaches separately for each task. For STS, we calculate the Pearson correlation between true similarity values against the predicted similarity values. For PARA and SST, we use accuracy as the metric since these are both classification tasks.

5.3 Experimental details

For all experiments, we standardize various hyperparmaters to keep the experiments constant which allows us to compare the methods more equally. Specifically, we use:

- **Batch size**: 64 (for all three tasks, SST, PARA, STS)
- **Optimizer**: AdamW optimizer with $1e-5$ learning rate
- **Individual losses**: for SST, we used cross entropy. For PARA, we used binary cross entropy with a sigmoid activation. For STS, we used Mean Squared Error (MSE).
- **Epochs**: 20. Experiments marked with a star were trained for 10 epochs[1]

Furthermore, we also experimented with two different data loading strategies. The first approach involved zipping together the three dataloaders for PARA, STS, and SST. However, zipping results in an iterable whose length is the shortest of the three dataloaders. Thus, we changed our approach to sample from each dataloader at each iteration of the epoch to yield the batches. One epoch is then defined as performing this n times where n is the length of the longest dataloader.

5.4 Results

In this section, we enumerate the experiment results of the methods described in section 4, as depicted in Table 1.

.

We tested each method described in section 4 on the dev set. We found that our proposed prototypical pre-training approach resulted in the best overall performance across the three tasks. We also

[1]Due to time and compute constraints.

[2]Experiments marked with a * denotes that this experiment was run for 10 epochs instead of the default 20 epochs due to time and compute constraints

Table 1: Dev set evaluations for all experiments and methods for each task and overall (average) [2].

Method	SST Acc (%)	PARA Acc (%)	STS Corr	Overall
Baseline	32.7	64.8	0.264	0.413
Unfrozen Baseline	49.0	78.2	0.383	0.552
Cosine Similarity STS	49.0	75.9	0.702	0.650
SST Weighted *	50.2	62.9	0.465	0.532
PARA Weighted *	32.1	72.7	0.369	0.472
STS Weighted *	29.6	63.9	0.571	0.502
Shared Layers *	47.9	69.4	0.259	0.477
Concatenation Before Encoding	49.6	80.5	0.833	0.711
Prototypical Pre-trained	49.0	**82.5**	**0.857**	**0.724**

Table 2: Test set evaluations for prototypical pre-trained method.

Method	SST Acc (%)	PARA Acc (%)	STS Corr	Overall
Prototypical Pre-trained	**51.3**	**82.2**	**0.849**	**0.728**

evaluated this approach on the test set and saw near identical overall performance (0.670 on the dev set and 0.672 on the test set).

In general, we found that the modifications to the baseline, described in section 4, generally improved the overall performance.

Overall, the sequential performance increases across the tested methods makes sense since each new method was constructed to explicitly add a new dimension to the solution or remedy a shortcoming of the previous methods tested. We further analyze the results and behavior of the methods in the section 6.

6 Analysis

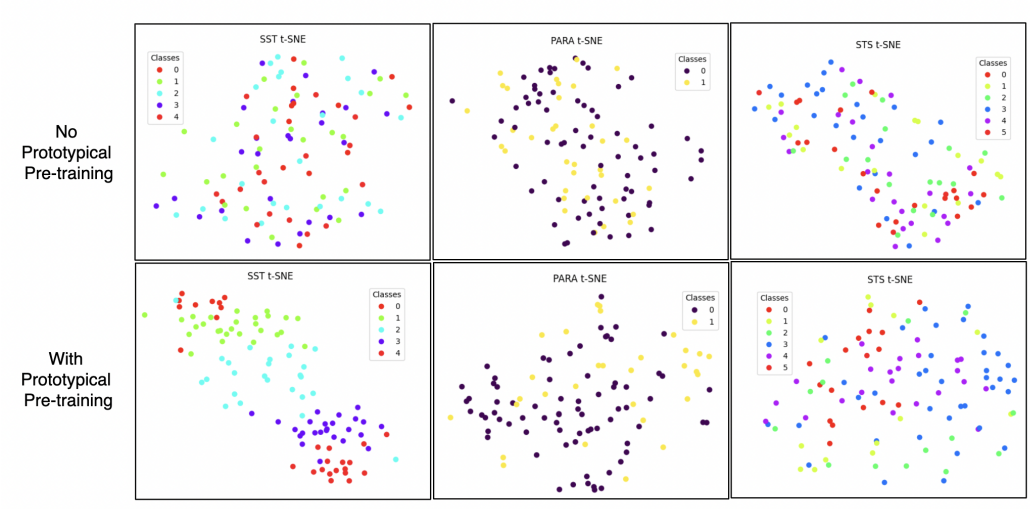

Figure 1: Class visualization using t-SNE (Van der Maaten and Hinton, 2008) embeddings of 100 samples for each task, before and after prototypical pre-training.

As we can see from the evaluation results, there were several ways to improve upon the supervised baseline. We see that keeping the parameters of f, the BERT backbone, during the course of fine-tuning improved upon the initial baseline which has the parameters of f frozen. This performance increase is expected in all methods since none of the methods were explicitly pre-trained on the given datasets except for the prototypical pre-trained experiment. However, even then, we empirically saw that the performance increased when keeping the parameters of f unfrozen. We hypothesize that pre-training, including prototypical pre-training, gives the entire model a more robust and accurate starting point for its parameters which enabled faster convergence and better results.

Next, was the use of cosine similarity instead of a linear MLP head for the STS task. We saw performance jump up from 0.383 on the unfrozen baseline to 0.702 on the STS task. Thus, for the remaining experiments, we chose to use the cosine similarity approach for STS over the linear MLP head. We hypothesize that the cosine similarity approach worked better than the linear head because cosine similarity explicitly tracks the "similarity" between vectors. Moreover, the linear MLP head does not have any explicit similarity inducing methods and thus, must learn which inputs are similar implicitly.

Another set of experiments we tried was weighting the individual losses to form the final weighted summation loss that we backpropogate with. As described in section 4, we tested three combinations where we weighted one of the three tasks much more heavily (by 98%) than the other tasks for each combination. As expected, we saw performance decrease in the tasks with low weight (0.01) and performance increase the in the heavily weighed task for each combination. For example, the STS weighted approach produced an STS correlation score of 0.571 which was higher than the STS correlation score in the other weigthed combinations. However, this same method produced a score of 29.6% on the SST task which was well below the other methods. This pattern held true across all combinations. We argue that this occurs due to destructive interference during training Yu et al. (2020). That is, if we weight one of the tasks more heavily, the optimization process will heavily favor that particular task and thus, destroy the gradients of the other tasks.

Another method we tried was sharing the linear layers between the PARA and STS MLP heads (s and p). Note that to enable this method, we use a linear MLP head for the STS task instead of the forementioned cosine similarity approach. Compared to the unfrozen baseline, we saw performance drop across all three tasks with this method. We hypothesize that this occurs because the parameters in the linear layers must now accommodate for learning useful representations for two separate tasks which makes the layers less robust. Thus, performance decreases. We also argue that, if the tasks were more similar to each other, then sharing layers can possibly improve performance.

Finally, we tried concatenating before encoding and found that this improved performance across the board. We hypothesize that this was due to cross-attention between the two inputs in PARA and STS.

Lastly, we tested our proposed prototypical pre-training method. We see that the average (overall) performance across all three tasks was the highest. Specifically, the method performed better than the both of the baselines and also the cosine similarity baseline. This method achieved the highest PARA accuracy and STS correlation and the second highest SST accuracy.

We hypothesize that, by explicitly learning a lower-dimensional feature space during prototypical pre-training, downstream fine-tuning of the overall model was more robust and performant. To further understand why this was the case, we decided to inspect the learned feature space across the three tasks using the t-SNE embedding visualization method Van der Maaten and Hinton (2008). Specifically, we encode 100 inputs using our prototypical pre-trained BERT backbone f to produce 100 latent vectors for each task. Each vector has a label associated with it denoting what class it belongs to. We then scatter plotted all of these vectors after t-SNE to produce a visualization of the learned latent space. This is shown in figure A.3.

Amazingly, as can be seen in the figure, the latent space for the SST task became almost linearly dividable by class. However, interestingly enough, this did not result in direct performance gains for the SST task, but the overall performance across the three tasks improved. We see a potential opportunity to improve performance by optimizing the number of additional information-rich dimensions we can re-anneal while maintaining this semblance of clustering. Alternatively, we can make use of UMAP graphs instead of t-SNE. This is increasingly preferred for making a fuzzy, but topologically representative graph of higher dimensions before compressing to lower dimensions and retaining higher-dimensional relationships, whereas t-SNE moves individual points from high-to-low

dimensional space. This visible clustering was not as evident for the STS and PARA tasks. We hypothesize that, during the jointly optimized pre-training phase, this method learned a *shared* feature latent space between all three tasks. Thus, we should not expect the performance to specifically increase in just one task but rather, that the generally learned feature space is more robust than the raw input space across multiple tasks. We hypothesize that this was the reason for why we saw a performance jump overall but not drastically in one specific task. We note that future work could be done to analyze the theoretical underpinnings to better explain the yielded results in more detail.

7 Conclusion

The goal of this project was to produce and evaluate methods for learning robust sentence embeddings that are performant across multiple tasks in downstream fine-tuning. Here, we successfully introduced the novel prototypical pre-training method for producing a robust learned feature space. Our proposed method saw performance gains overall, compared to several baselines and all of the other methods tested.

We highlight that there is a need to further analyze why prototypical pre-training works the way it does. Future work would include such an analysis, as well as testing different variations of prototypical pre-training. Such variations may include weighted ensembling during the pre-training and/or fine-tuning phase or computing the prototypes on a per-epoch basis rather than a per-batch basis. Overall, the main contribution of this project is introducing a novel pre-training method for multi-task learning which can be built upon and further analyzed to improve results in the future.

References

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-sne. *Journal of machine learning research*, 9(11).

Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical networks for few-shot learning. *Advances in neural information processing systems*, 30.

Mike Wu, Noah Goodman, Chris Piech, and Chelsea Finn. 2021. Prototransformer: A meta-learning approach to providing student feedback. *arXiv preprint arXiv:2107.14035*.

Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. 2020. Gradient surgery for multi-task learning. *Advances in Neural Information Processing Systems*, 33:5824–5836.

A Appendix

A.1 Model details

A.1.1 MLP Architectures

For the supervised baseline described in section 4.1, we constructed three multi-layer perceptron heads for PARA (p), STS (s), SST (t). Each of three MLP head layers are denoted in the following tables.

Number	Layer	PyTorch Code
1	Dropout ($p = 0.3$)	`nn.Dropout(0.3)`
2	Linear	`nn.Linear(BERT_HIDDEN_SIZE, N_SENTIMENT_CLASSES)`

Table 3: SST (t) MLP head architecture.

Number	Layer	PyTorch Code
1	Dropout Sentence 1 ($p = 0.3$)	`nn.Dropout(0.3)`
1	Dropout Sentence 2 ($p = 0.3$)	`nn.Dropout(0.3)`
2	Linear Sentence 1	`nn.Linear(BERT_HIDDEN_SIZE, BERT_HIDDEN_SIZE)`
2	Linear Sentence 2	`nn.Linear(BERT_HIDDEN_SIZE, BERT_HIDDEN_SIZE)`
3	Mult	`torch.mul(Sentence 1, Sentence 2)`
4	Linear	`nn.Linear(BERT_HIDDEN_SIZE, 1)`

Table 4: PARA (p) MLP head architecture.

Number	Layer	PyTorch Code
1	Dropout Sentence 1 ($p = 0.3$)	`nn.Dropout(0.3)`
1	Dropout Sentence 2 ($p = 0.3$)	`nn.Dropout(0.3)`
2	Linear Sentence 1	`nn.Linear(BERT_HIDDEN_SIZE, BERT_HIDDEN_SIZE)`
2	Linear Sentence 2	`nn.Linear(BERT_HIDDEN_SIZE, BERT_HIDDEN_SIZE)`
3	Mult	`torch.mul(Sentence 1, Sentence 2)`
4	Linear	`nn.Linear(BERT_HIDDEN_SIZE, 1)`

Table 5: STS (s) MLP head architecture.

A.2 Prototypical Step Pseudocode

The following is code for producing the logits from the inputs using prototypes (i.e., the prototypical "step"), as described in section 4.

```
def proto_step(self, input_ids, attention_mask, labels):
        """
        Runs the protonet approach to produce the logits for
        the given input for the given task.
        """

        # only do this for "classification" tasks
        assert labels.dtype == torch.int64

        # embed into feature space (will be of shape [batch_size, hidden_size])
        latents = self.encode(input_ids, attention_mask)

        # note that if sentence pair is input, we'd do combine them first:
        latents = self.encode(input_ids_1, attention_mask_1) +
```

```
        self.encode(input_ids_2, attention_mask_2)

        # compute class vectors
        label_map = self.create_label_map(labels)
        new_labels = self.transform_labels(labels, label_map)
        class_vectors = self.create_class_vectors(latents, new_labels)


        # get mean for each class
        prototypes = {}
        for key in class_vectors:
            prototypes[key] = torch.mean(torch.stack(class_vectors[key]), dim=0)

        proto_list = [prototypes[key] for key in sorted(prototypes.keys())]
        protos = torch.stack(proto_list)

        # move protos and new_labels to device
        protos = protos.to(self.device)
        new_labels = new_labels.to(self.device)

        logits = -torch.cdist(latents, protos)
        loss_val = F.cross_entropy(logits, new_labels)
        accuracy_val = self.calculate_score(logits, new_labels)

        return loss_val, accuracy_val
```

A.3 Prototypical Pre-training and Fine-tuning Diagram

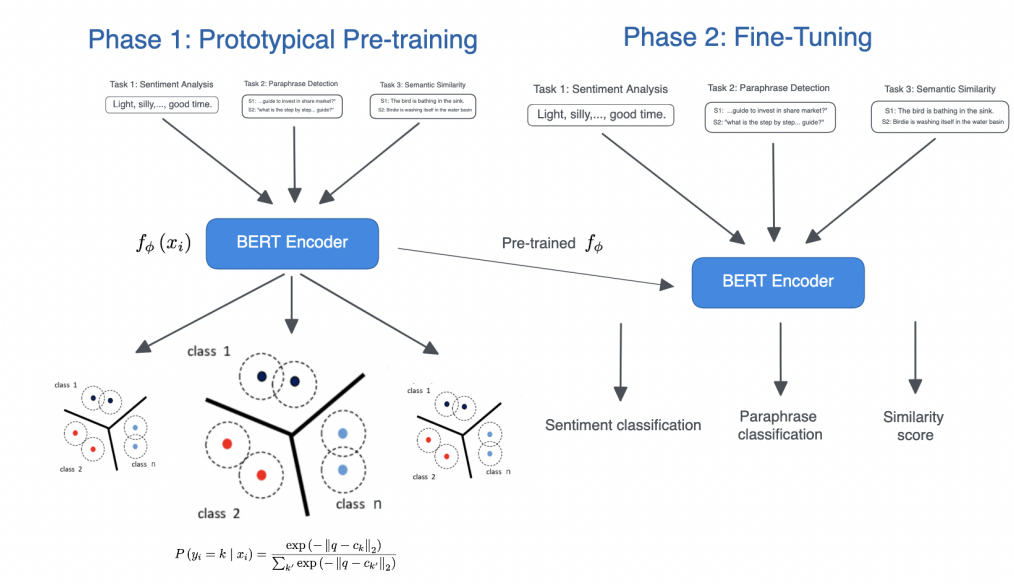

Figure 2: Overview of the prototypical pre-training and fine-tuning approach described in section 4.