vat-pytorch: A Plug-and-Play library for Virtual Adversarial Training

Kevin Blin
keblin@student.ethz.ch

Flavio Schneider
scflavio@student.ethz.ch

Steven H. Wang
stewang@student.ethz.ch

Abstract

Virtual Adversarial Training (VAT) is an adversarial regularization technique that is commonly used when fine-tuning large language models. We release a plug-and-play pip package, `vat-pytorch`[1], which provides easy-to-use implementations of three VAT algorithms, SMART, ALICE, and ALICE++. To the best of our knowledge, `vat-pytorch` provides the first public implementations of ALICE and ALICE++. We carefully benchmark the performance of our implementations[2] on two temporal commonsense-reasoning classification datasets following (Pereira et al., 2021), and show that in the MCTACO benchmarks, using our VAT implementations improves benchmark scores compared to fine-tuned and AdamW weight decay baselines. Finally, we attempt a transfer of ALICE to the extractive question answering setting for a legal NLP dataset and report on the results of preliminary experiments.

1 Introduction

Fine-tuning pretrained models is a common task in many fields of machine learning. This is especially true in Natural Language Processing, where many state-of-the-art methods start from fine-tuning large, pretrained Transformer-based language models like BERT (Devlin et al., 2019).

Typically the training dataset used for fine-tuning language models is much smaller than initial dataset, leading to a significant risk of overfitting. Standard techniques for combatting overfitting include data augmentation techniques, which have seen great success in image classification, but are considered more challenging to use for text data (Wang et al., 2017; Feng et al., 2021), model complexity reduction techniques which freezing most parameters in the network or train a smaller number of new parameters as in Adapter modules (Houlsby et al., 2019), and regularization techniques like Dropout or weight decay (Srivastava et al., 2014; Loshchilov and Hutter, 2017).

[1]Code is available at https://github.com/archinetai/vat-pytorch.

[2]Benchmark code can be found in a separate repository at https://github.com/TheBlueHawk/CS4NLP_Project2022.

2 Virtual Adversarial Training

Virtual Adversarial Training (VAT) is a regularization technique that penalizes sudden changes in model outputs. Thus, VAT incentivizes local distributional smoothness (Miyato et al., 2015, 2019).

Several state-of-the-art models in NLP have used VAT techniques to combat overfitting during the fine-tuning of large language models. One example is KEAR (Xu et al., 2021) which achieved human parity on the CommonSenseQA benchmark (Talmor et al., 2018).

VAT adds to the training objective a regularization term that penalizes an f-divergence (e.g. KL divergence) between the model output corresponding to the original input and the model output corresponding to a perturbed input, which is adversarially chosen to maximize this penalty (Miyato et al., 2019). The adversarial perturbations to the input are constrained inside an l_p ball.

VAT is similar to standard adversarial training, but unlike standard adversarial training can be applied to semi-supervised settings (it relies only on the model outputs instead of ground-truth outputs), and is generally less computationally expensive.

2.1 VAT applied to common-sense reasoning tasks in NLP

SMART, ALICE, and ALICE++ are recent NLP-focused VAT techniques shown to improve transfer learning for large language models, and which broke SOTA records on common-sense reasoning benchmarks like GLUE (Wang et al., 2018) and MCTACO (Zhou et al., 2019).

SMART SMART (Jiang et al., 2020) was the first application of VAT to the fine-tuning of pre-trained language models, and at time of publication set a new state-of-the-art for the GLUE commonsense reasoning benchmark.

SMART adds an adversarial regularization term $\mathcal{R}_{\mathrm{S}}$ to the loss function. SMART uses projected gradient descent to optimize an adversarial perturbation δ applied to the token embeddings x (which are the inputs to the fine-tuned model). The perturbation δ is restricted to an l_p ball of radius ϵ, and is chosen to maximize the symmetrized KL divergence ℓ_{S} between the model output and a perturbed model output.

$$\mathcal{R}_{\mathrm{S}}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \max_{\delta} \ell_{\mathrm{S}}\left(f_\theta\left(x_i+\delta\right), f_\theta\left(x_i\right)\right) \tag{1}$$

SMART also uses Bregman Proximal Point Optimization to increase stability during training, but the performance gain due to Bregman Proximal Point Optimization is insignificant compared to the gains due to VAT.

ALICE In ALICE (Pereira et al., 2020), the language model's loss function is replaced by $\mathcal{L}_{\mathrm{ALICE}}$, a the sum of two adversarially optimized terms based on the original loss function l.

$$\begin{aligned}\mathcal{L}_{\mathrm{ALICE}}(\theta) = \sum_{i=1}^{N} [&\max_{\delta_1} l\left(f_\theta(x_i+\delta_1), y_i\right) \\ &+ \alpha \max_{\delta_2} l\left(f_\theta\left(x_i+\delta_2\right), f_\theta(x_i)\right)]\end{aligned} \tag{2}$$

As in SMART, the perturbations are applied to the token embeddings, and are restricted to an l_p ball. The first perturbation δ_1 is a standard adversarial optimization against the ground-truth labels y. The second perturbation δ_2 (similar to the perturbation in SMART) is an adversarial optimization against the model outputs, or "virtual labels."

Before June 2022, ALICE topped the AllenAI leaderboard for the MCTACO benchmark. Pereira et al. (2020) report that ALICE exceeds the performance of SMART on three benchmarks, but do not compare SMART to ALICE on GLUE. Instead they report performance on COSMOSQA (Huang et al., 2019), which seems similar to GLUE in task and size.

ALICE++ ALICE++ (Pereira et al., 2021) is an extension to ALICE. It adds to ALICE a hyperparameter K (where $0 \leq K \leq L$ and L is the number of layers in the network). At the start of every update ALICE++ uniformly randomly chooses a layer $0, 1, .., K$ (layer 0 is defined as the input embedding "layer") as the target of adversarial perturbations. ALICE++ with $k = 0$ only perturbs the the input embedding layer, and is therefore equivalent to ALICE.

Pereira et al. (2021) also explore replacing the original loss function l in Equation 2 with f-divergences like the Jensen-Shannon Divergence, which is found to improve performance on several benchmarks.

3 `vat-pytorch` design and implementation details

We release our Virtual Adversarial Training implementations as the `pip` package `vat-pytorch`[3], designed to be used with HuggingFace Transformers (Wolf et al., 2020), but which in principle can be used with any pretrained language models.

Earlier this year, we also released a standalone SMART-only package `smart-pytorch`, which currently has over 30 GitHub stars.

Challenges A main challenge with the implementation of virtual adversarial methods is that the model must be executed in different chunks, and different gradient settings depending on whether we are optimizing the adversarial noise or the model itself. This is already enough to break the HuggingFace Transformers abstractions. Furthermore, noise must be applied to the embedding inputs or intermediate outputs of the network and the resulting perturbation must be inspected on the logits. Different architectures have different structures and different amount of layers, making a generalized design nonobvious.

Overall Design To overcome these challenges, `vat-pytorch` provides an API where the user can provide an `ExtractedModel`, a wrapped version of a Huggingface Transformers model which calculate the model's output logits given the input to any layer of the model. The `ExtractedModel` is internally called by our library with the perturbed values to produce VAT loss.

Since often the user is interested in obtaining the output logits together with the loss, at least one complete execution of the model is necessary. In order to avoid additional time-consuming calls by

[3] https://github.com/archinetai/vat-pytorch

```python
import torch
import pytorch_vat as vat
from transformers import AutoTokenizer

extracted_model = vat.models.ExtractedRoBERTa()
tokenizer = AutoTokenizer.from_pretrained('roberta-base')
# Pick one:
model = vat.models.SMARTClassificationModel(extracted_model)
model = vat.models.ALICEClassificationModel(extracted_model)
model = vat.models.ALICEPPClassificationModel(extracted_model)
# Compute inputs
text = ["This text belongs to class 1...", "This text belongs to class 0..."]
inputs = tokenizer(text, return_tensors='pt')
labels = torch.tensor([1, 0])
# Compute logits and loss
logits, loss = model(
    input_ids=inputs['input_ids'],
    attention_mask=inputs['attention_mask'],
    labels=labels)
# To finetune do this for many steps
loss.backward()
```

Figure 1: Example code demonstrating how to use `vat-pytorch` for a classification task. `vat-pytorch` provides *high-level* plug-and-play model wrappers that automatically apply adversarial perturbations during training to any pretrained model that implements the `ExtractedModel` interface. We also provide *low-level* implementations of each VAT algorithm for more detailed use cases.

our library, the embedding and state computed on the first forward pass (used by the user to obtain the logits) must be provided as input to our virtual loss. In the case of ALICE++ instead of the embedding, all the intermediate hidden states must be provided.

As example of an `ExtractedModel` implementation for a particular model, `ExtractedRoBERTa` can be found in the appendix (Figure 7) and README of the library.

Overviews of the `vat-pytorch` APIs and implementation details for each VAT algorithm follow:

3.1 SMART

The original implementation of SMART is available in the MT-DNN[4] repository. However, this implementation is mired in overlapping research code, as the repository is intended for studying multi-task training. In `vat-pytorch` we provide a more accessible and easy-installed implementation of SMART, `SMARTLoss`.

For simplicity, we implement only the VAT part, and not the Bregman Proximal Optimization part of SMART since ablation studies in (Jiang et al., 2020) show its effect on model performance to be very minor despite its computational cost.

`SMARTLoss(.)` has the following parameters (1) `model` the `ExtractedModel` to be perturbed (2) `loss_fn` the loss function applied between the output state and the target state at each iteration (3) `loss_last_fn` the loss function applied on the last iteration (4) `norm_fn` the normalization used on the perturbation ball (5) `num_steps` the number of optimization steps (6) `step_size` the noise gradient step-size (7) `noise_var` the variance of the perturbation noise.

3.2 ALICE

The implementation of ALICE (`ALICELoss`) uses twice the same adversarial perturbation loop as SMART, but with different parameters. The first is analogous to SMART where we perturb the text embedding with the goal of influencing the output state of the model $f(x;\theta)$, and regularize to make sure that the perturbation is minimal. The second additional perturbation is again applied on the embeddings, but this time the output labels y of the model are regularized.

[4] https://github.com/namisan/mt-dnn

Measure		Value
# of unique questions		1893
# of unique question-answer pairs		13,225
Category	# questions	avg # of cand.
event frequency	433	8.5
event duration	440	9.4
event stationarity	279	3.1
event ordering	370	5.4
event typical time	371	6.8

Table 1: Basic statistics for the MCTACO dataset, including the number of examples of each question category. Only a quarter of the dataset is publicly available for download – the rest is reserved as the test set for leaderboard rankings.

In addition to the SMART parameters the number of labels `num_classes` must be provided, `gold_loss_fn` and `gold_loss_last_fn` can be selected to differentiate the loss on the labels, and `alpha` can be selected to change the weighting between the two losses.

3.3 ALICE++

The implementation of ALICE++ (`ALICEPPLoss`) borrows most of the code from ALICE, but at each call, a different layer of the model is perturbed. The challenge with this implementation is that to perturb an intermediate layer, we must be able to run the sub-model that goes from layer i to the last layer of the network. Other than ALICE hyperparameters, the number of total perturbable layers `num_layers` must be selected and the maximum perturbable layer `max_layer` can be selected.

4 Temporal Reasoning Datasets

Following Pereira et al. (2021), we carefully benchmark our VAT implementations on the temporal reasoning tasks MCTACO and TimeML, which we introduce in this section.

4.1 MCTACO: Multiple Choice TemporAl COmmensense Dataset

The Multiple Choice TemporAl COmmonsense (MCTACO) dataset (Zhou et al., 2019) consists of 1893 multiple choice temporal commonsense reasoning questions, with 13,225 unique question-answer pairs.

There are 5 different time-related categories of questions in MCTACO: frequency, duration, stationarity, order, and typical time. Every tuple in the dataset is structured as follows: There is a single context sentence (such as "There were a couple times that the family dog, Mika, has tried to take Joey from Marsha and eat him!"). Then there is a temporal reasoning question (e.g. "How long does it take Mika to try to eat Joey?"). Finally, there is a set of possible answers (e.g. "1 hour", "1 century", "under a year"). More than one answer can be true.

This dataset is evaluated on the exact match (EM) and macro-average F1 scores. Human performance achieves an F1 score of 87.1 and an EM score of 75.8. For example, in the sentence “Mary **visited** the grocery store." the event “visited" takes less than a day.

ALICE reported an F1 score of 79.50 and EM score 56.45 on the test set. ALICE++ reported an F1 score of 80.00 and an EM score of 59.90 on the test set.

4.2 TimeML

TimeML (Saurí et al., 2006) is a dataset of temporal commonsense reasoning annotations, including annotations indicating the temporal order, causal relation, and durations of events in a sentence.

Following (Pereira et al., 2021), we use an event duration prediction task extracted from TimeBank where the model considers whether some event in a sentence has a duration longer or shorter than a day. The extracted dataset contains 2,251 examples, split into train and test sets of 1248 and 1003 examples each.

5 Experiments

We describe our benchmarking methodology, including our model selection procedure, and compare `vat-pytorch`'s performance against results reported by Pereira et al. (2020) and Pereira et al. (2021).

5.1 Methodology

For both datasets we compare 4 models: a strong baseline and our implementations of SMART, ALICE, ALICE++. Each of these models is the results of a careful hyperparameter optimization on the development set that we divided in a train set and a validation set. Unless stated otherwise, the standard procedure we used is as follows:

1. We reserved 10% of the training dataset as a validation set during the hyperparameter search.

2. Using the Weights and Biases ML experiment tracking platform (Biewald, 2020), we conducted a random search or Bayesian search[5] over a large range of values centered around the best hyperparameters reported in the literature for these models.

3. We performed a narrow grid search sweep centered around the most promising regions obtained from the random search. Each possible combination of hyperparameters is tested over 5 fixed seeds, and then their performance is averaged.

4. Once the best combination of hyperparameters is determined by the grid search, we final evaluate the model on the test set with 10 runs.

This procedure allowed us to produce robust and reproducible results without leaking any data from the final test set during the model training and validation. More implementations details common to all models (unless stated otherwise), include the use of the ADAMW (Loshchilov and Hutter, 2017) optimizer with default weight decay of 0.01, [6] 20 epochs of training, the pretrained model is `roberta-base` with batch size of 16, and the runtime is based on a single Nvidia RTX2080Ti GPU. These choice of pretrained model and batch size were limited by the computational resources available. To conserve GPU memory we used 16-bit precision and gradient accumulation.

5.2 MCTACO

5.2.1 Baselines

Our set of baselines on the MCTACO dataset are a selection of popular Transformer-based language models (`bert-base-uncased`, `roberta-base`, and `roberta-large`).

[5]Our early experiments used random search, but we switched to Bayesian search later to conserve computational resources

[6]We had intended to isolate ADAMW's regularization effect from VAT regularization to compare these two techniques, but mistakenly forgot to turn off weight decay. However, as we explain later, ADAMW was shown to have no effect on training except for with very large weight decay hyperparameter settings, so this should have minimal consequence for our MCTACO results.

Vanilla roberta-base baseline As a baseline for our VAT algorithms, we fine-tune vanilla `roberta-base` on MCTACO. We performed hyperparameter optimization over their learning rates, see Appendix A for more details.

ADAMW weight decay baseline To compare out VAT implementations against a different regularization technique, we also fine-tuned `roberta-base` with ADAMW weight decay. However, we found in preliminary experiments that performance was unchanged for small values of weight decay, and was only harmed by large values of weight decay.[7] Therefore, we omitted ADAMW from our final experimental results.

The performance of these baseline models is shown in Table 2, and matches the results reported in (Zhou et al., 2019).

5.2.2 VAT results

Due to limited computational resources, we studied the effectiveness of our VAT implementations exclusively on `roberta-base` as it is a modern architecture but relatively small. Pereira et al. (2021) however report on `roberta-large` results.

We find that all our VAT models significantly outperform the roberta-base baseline, but their scores within range of error of each other's performance.

VAT methods introduce a whole new set of hyperparameters linked to the adversarial perturbation that need to be tuned (radius, variance, epsilon) and algorithm-specific parameters like step-size for the inner optimization and weight coefficient for the VAT loss term. Search ranges as well as our final hyperparameters can be found in Appendix A.

Model	EM	F1
`roberta-base`	44.3 ± 1.1	69.7 ± 0.9
`SMART`	$\mathbf{48.3} \pm 0.9$	72.4 ± 1.3
`ALICE`	47.5 ± 0.9	72.2 ± 1.2
`ALICE++`	47.9 ± 0.9	$\mathbf{73.3} \pm 0.9$

Table 2: Mean and standard errors for MCTACO scores on our vanilla baseline and different hypertuned models, averaged over 10 runs. VAT models are based on `roberta-base`.

SMART As we can see in image 3, SMART significantly reduces the model overfitting and improves the model performance under every metric. Given the training curve, it could be even argued

[7]See this Weights and Biases report for details.

that a higher number of epochs could have lead to even better results.

ALICE Surprisingly, ALICE performs very similarly to SMART and doesn't bring any particular improvements, and its accuracy score (shown in Figure 2) is lower. This is in stark contast with the literature and despite our best efforts we did not manage to achieve significantly higher results.

ALICE++ ALICE++ performs significantly better than the other models, quickly reaching its top performance and setting the highest F1. SMART achieves a slightly high score but only after more than twice the number of training epochs.

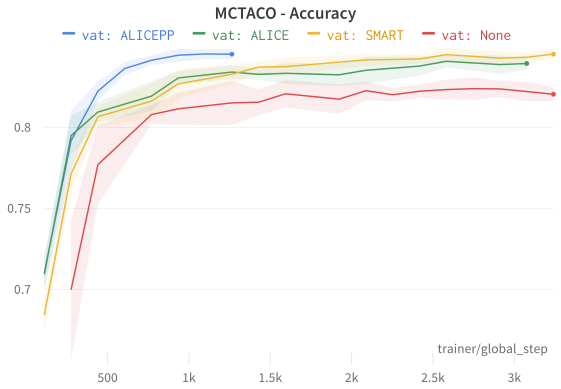

Figure 2: Accuracy of `roberta-base` fine-tuned on MCTACO using different VAT methods.

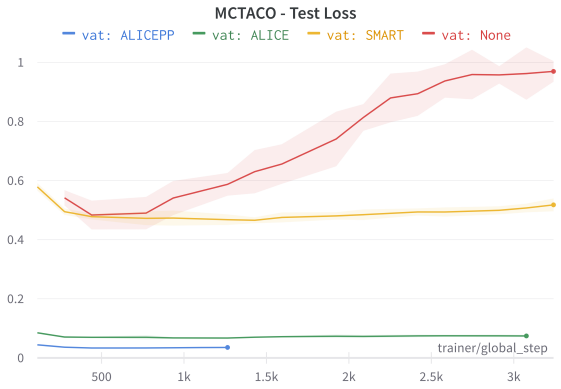

Figure 3: Test loss of `roberta-base` fine-tuned on MCTACO using different VAT methods. Note that each curve uses a different loss function. However, we observe a general pattern where test loss increases most dramatically for the baseline run, suggesting that only the baseline run is overfitting.

5.2.3 ALICE++ and `roberta-large`

After achieving strong results with ALICE++ and `roberta-base`, we benchmarked our implementation on the more computationally expensive `roberta-large` model, which allows for an apples-to-apples comparison against the original ALICE++ results in (Pereira et al., 2021). As shown in Table 3, our ALICE++, although improving over our VAT+roberta-base results, has lower performance than those reported in (Pereira et al., 2021). Perhaps this could be due to several engineering tricks that we didn't had the time to implement like dropout, gradient clipping, learning rate decay and warmup.

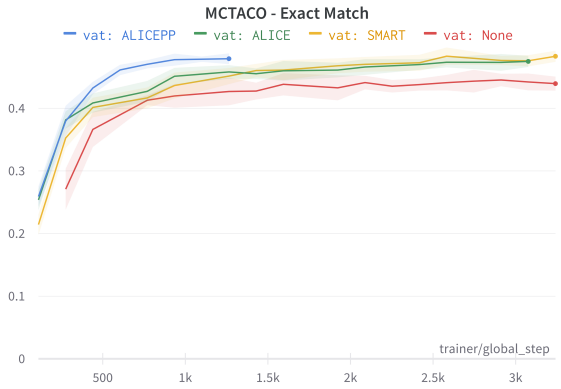

Figure 4: Exact-Match score of `roberta-base` fine-tuned on MCTACO using different VAT methods.

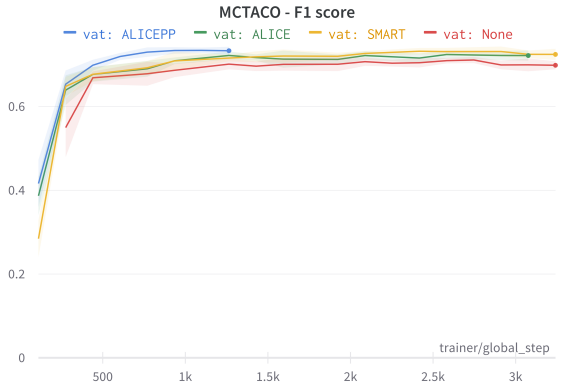

Figure 5: F1 score of `roberta-base` fine-tuned on MCTACO using different VAT methods.

Model	EM	F1
`roberta-large`	47.5	72.4
`ALICE++` (Ours)	52.3	77.2
`ALICE++` (Original)	**58.1**	**80.2**

Table 3: MCTACO score comparison of our ALICE++ implementation on roberta-large versus the original results reported in Pereira et al. (2021) and a vanilla baseline. In both cases of ALICE++ no best layer selection is performed.

5.3 TimeML dataset

We benchmark a `roberta-base` baseline and our three VAT algorithms against the TimeML benchmark with the same methodology as with MCTACO.

Model	Accuracy
`roberta-base`	83.1 ± 1.2
`SMART`	83.7 ± 0.9
`ALICE`	83.3 ± 0.8
`ALICE++`	83.6 ± 0.7

Table 4: TimeML accuracy scores of different hyper-tuned models, averaged over 10 runs. VAT models are based on `roberta-base`. We observe that all scores fall within range of error of the baseline, indicating that our VAT algorithms did not improve the model performance.

Despite our VAT algorithms creating a more test loss curves that indicate less overfitting (Figure 6), we do not see best test accuracy as shown in Table 4, with the test accuracy of every VAT model falling within range of error of the vanilla baseline.

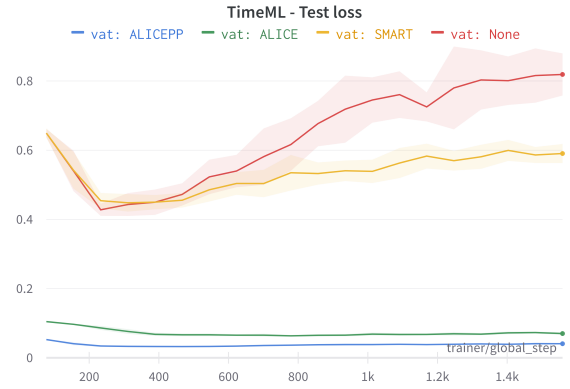

Figure 6: Test loss of `roberta-base` fine-tuned on TimeML using different VAT methods.

6 ALICE for Extractive QA

We conducted a preliminary transfer of our ALICE implementation to the Extractive QA setting on a legal dataset.

6.1 CUAD: The Contract Understanding Atticus Dataset

The Contract Understanding Atticus Dataset (CUAD) (Hendrycks et al., 2021) is an Extractive Question Answering benchmark for contract review, an specialized legal NLP task of identifying key contractual obligations inside legal contracts.

The CUAD dataset consists of 510 full-length contracts of various types (e.g. IP Agreement, Sponsorship Agreement) and over 13,000 "highlighted" annotations of 41 different types of important legal clauses. Models trained on the CUAD dataset aim to identify, for each of the 41 different legal clauses, the span of the corresponding highlighted contract text if it exists in the contract.

VAT could be a good fit for CUAD because the dataset is highly imbalanced. Less than 1% of the data consists of positive examples, the rest being text that is irrelevant for the task. (This sort of data imbalance with rare positive examples or rare examples of certain categories in common in many legal NLP tasks.) Therefore adversarial training could serve as a indirect data augmentation method for the rare positive examples. [8]

Since this dataset is highly unbalanced, CUAD uses the area under the precision-recall curve (AUPR) as a primary performance metric. The best performing fine-tuned baseline models by this metric are `DeBERTa-large`, with AUPR 47.2 and `RoBERTa-large`, which has AUPR 48.2.

6.2 Adapting ALICE to the Extractive Question Answering setting

In Pereira et al. (2020) and Pereira et al. (2021), ALICE was benchmarked on classification tasks only, though in principle ALICE can be applied to any finetuning task.

Devlin et al. (2019) introduced a simple Extractive QA technique wherein BERT sweeps through the text in windows and generates start and end logits for every token in the the window, by taking the dot product between the final hidden vector of each token and two trainable vectors S and T (the start and end vectors). The same approach is used for all fine-tuned Transformer-based baselines in Hendrycks et al. (2021).

To adapt ALICE to the Extractive QA setting, we simply add a second loss term to each adversarial optimization in Equation 2, leading to the following loss function:

$$\mathcal{L}_{\text{ALICEQA}}(\theta) = \sum_{i=1}^{N} \max_{\delta_1} L_1(\delta_1) + L_2(\delta_1) + \max_{\delta_2} L_1(\delta_2) + L_2(\delta_2) \quad (3)$$

[8] Steven also has personal curiosity for this task because he is working with The Atticus Project. Otherwise, we could have tried SQUAD.

where

$$L_1(\delta) = l(f_{\theta,\text{start}}(x_i + \delta), y_{\text{start},i})$$

$$L_2(\delta) = l(f_{\theta,\text{end}}(x_i + \delta), y_{\text{end},i})$$

Instead of choosing an adversarial perturbation δ optimized for the divergence between a single perturbed output and the corresponding label (or virtual label), we choose an adversarial perturbation that maximizes the sum of the divergence between the start logit and its (virtual) label and the divergence between the end logit and its (virtual) label.

6.3 Extractive QA Experimental Results

In our preliminary experiment, we evaluate the test set performance of `roberta-base` with ALICE when trained on the full training set without validation, using the default hyperparameters for `roberta-base` provided in the CUAD GitHub repository [9].

We compare against the vanilla `roberta-base` performance reported in Hendrycks et al. (2021) and also compare against a baseline regularizer, AdamW, using weight decay hyperparameters 1e-3, 1e-2, and 1e-1. We report the average over three runs of the best weight decay hyperparameter, 1e-2.

Model	AUPR
`roberta-base`	0.426
ALICE	0.378
AdamW	0.458

Table 5: Average AUPR scores for our preliminary CUAD experiments. Results averaged over 3 runs for the AdamW baseline, and over 2 runs for ALICE.

We fine that our ALICE for Extractive QA implementation hurts performance, suggesting that we may need to do more hyperparameter tuning or review the implementation. Our implementation and training scripts can be found at github.com/shwang/CUAD.

7 Conclusion

We present `vat-pytorch`, a library for virtual adversarial training algorithms, including (to the best of our knowledge) the first public implementations of ALICE and ALICE++.

[9] github.com/TheAtticusProject/CUAD

We benchmark the performance of our implementations on two temporal commonsense reasoning sets, following Pereira et al. (2020) and Pereira et al. (2021). Due to computational constraints, we conducted most of our experiments on `roberta-base` instead of `roberta-large`. We find that for the MCTACO benchmark, our VAT improve the F1 and EM scores for `roberta-base` over the vanilla and weight-decay baselines. However, our `roberta-large` experiments do not meet previously reported scores.

We hope that the release of `vat-pytorch`, will allow for easy usage and wider adoption of these VAT techniques in NLP. Indeed, an earlier iteration of our library `smart-pytorch` already has over 30 stars on GitHub.

References

Lukas Biewald. 2020. Experiment tracking with weights and biases. Software available from wandb.com.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for NLP. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988, Online. Association for Computational Linguistics.

Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. 2021. Cuad: An expert-annotated nlp dataset for legal contract review. *ArXiv*, abs/2103.06268.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR.

Lifu Huang, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Cosmos qa: Machine reading comprehension with contextual commonsense reasoning. 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)*, pages 2391–2401.

Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. 2020. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In *ACL*.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization.

Takeru Miyato, Shin ichi Maeda, Masanori Koyama, and Shin Ishii. 2019. Virtual adversarial training: A regularization method for supervised and semi-supervised learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41:1979–1993.

Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, Ken Nakae, and Shin Ishii. 2015. Distributional smoothing with virtual adversarial training.

Lis Pereira, Fei Cheng, Masayuki Asahara, and Ichiro Kobayashi. 2021. Alice++: Adversarial training for robust and effective temporal reasoning. In *PACLIC*.

Lis Pereira, Xiaodong Liu, Fei Cheng, Masayuki Asahara, and Ichiro Kobayashi. 2020. Adversarial training for commonsense inference. *ArXiv*, abs/2005.08156.

Roser Saurí, Jessica Littman, Bob Knippen, Robert Gaizauskas, Andrea Setzer, and James Pustejovsky. 2006. Timeml annotation guidelines. *Version*, 1(1):31.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. 2018. Commonsenseqa: A question answering challenge targeting commonsense knowledge.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Jason Wang, Luis Perez, et al. 2017. The effectiveness of data augmentation in image classification using deep learning. *Convolutional Neural Networks Vis. Recognit*, 11:1–8.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yichong Xu, Chenguang Zhu, Shuohang Wang, Siqi Sun, Hao Cheng, Xiaodong Liu, Jianfeng Gao, Pengcheng He, Michael Zeng, and Xuedong Huang. 2021. Human parity on commonsenseqa: Augmenting self-attention with external attention. *CoRR*, abs/2112.03254.

Ben Zhou, Daniel Khashabi, Qiang Ning, and Dan Roth. 2019. "going on a vacation" takes longer than "going for a walk": A study of temporal commonsense understanding. *ArXiv*, abs/1909.03065.

A Best Hyperparameter Details

Final hyperparameters can be found on our github repository[10] in the respective `final_<vat>_<dataset>.yaml` files.

[10] https://github.com/TheBlueHawk/CS4NLP_Project2022

```python
import torch.nn as nn
from transformers import AutoModelForSequenceClassification
from vat_pytorch.models import ExtractedModel

class ExtractedRoBERTa(ExtractedModel):

    def __init__(self):
        super().__init__()
        model = AutoModelForSequenceClassification.from_pretrained('roberta-base')
        self.roberta = model.roberta
        self.layers = model.roberta.encoder.layer
        self.classifier = model.classifier
        self.attention_mask = None
        self.num_layers = len(self.layers) - 1

    def forward(self, hidden, with_hidden_states = False, start_layer = 0):
        """ Forwards the hidden value from self.start_layer layer to the logits. """
        hidden_states = [hidden]

        for layer in self.layers[start_layer:]:
            hidden = layer(hidden, attention_mask = self.attention_mask)[0]
            hidden_states += [hidden]

        logits = self.classifier(hidden)

        return (logits, hidden_states) if with_hidden_states else logits

    def get_embeddings(self, input_ids):
        """ Computes first embedding layer given inputs_ids """
        return self.roberta.embeddings(input_ids)

    def set_attention_mask(self, attention_mask):
        """ Sets the correct mask on all subsequent forward passes """
        self.attention_mask = self.roberta.get_extended_attention_mask(
            attention_mask,
            input_shape = attention_mask.shape,
            device = attention_mask.device
        ) # (b, 1, 1, s)
```

Figure 7: `ExtractedRoBERTA`, an example of a pretrained model fitting the `ExtractedModel` interface for the `vat-pytorch` library.