A Greedy Subset Selection Algorithms

This section briefly reviews the technical details of greedy subset selection algorithms used in our experiments. Further details can be found in [27,16].

A.1 CRAIG

As discussed previously, the goal of coreset selection is to find a subset of the training data such that the weighted gradient computed over this subset can give a good approximation to the full gradient. Thus, CRAIG [27] starts with explicitly writing down this objective as:

$$\arg\min_{S\subseteq V} |S| \text{ s.t. } \max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \left\| \sum_{i\in V} \nabla_{\boldsymbol{\theta}} \boldsymbol{\Phi}\left(\boldsymbol{x}_i, y_i; f_{\boldsymbol{\theta}}\right) - \sum_{j\in S} \gamma_j \nabla_{\boldsymbol{\theta}} \boldsymbol{\Phi}\left(\boldsymbol{x}_j, y_j; f_{\boldsymbol{\theta}}\right) \right\| \leq \epsilon. \quad (16)$$

Here, $V = [n] = \{1, 2, \ldots, n\}$ denotes the training set. The goal is to find a coreset $S \subseteq V$ and its associated weights γ_j such that the objective of Eq. (16) is minimized. To this end, Mirzasoleiman *et al.* [27] find an upper-bound on the gradient estimation error of Eq. (16). This way, it is shown that the coreset selection objective can be approximated by:

$$S^* = \arg\min_{S\subseteq V} |S|, \quad \text{s.t.} \quad L(S) \triangleq \sum_{i\in V} \min_{j\in S} d_{ij} \leq \epsilon, \quad (17)$$

where

$$d_{ij} \triangleq \max_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \left\|\nabla_{\boldsymbol{\theta}} \boldsymbol{\Phi}\left(\boldsymbol{x}_i, y_i; f_{\boldsymbol{\theta}}\right) - \nabla_{\boldsymbol{\theta}} \boldsymbol{\Phi}\left(\boldsymbol{x}_j, y_j; f_{\boldsymbol{\theta}}\right)\right\| \quad (18)$$

denotes the maximum pairwise gradient distances computed for all $i \in V$ and $j \in S$. Then, Mirzasoleiman *et al.* [27] cast Eq. (17) as the well-known *submodular set cover problem* for which greedy solvers exist [26,29,40].

A.2 GradMatch

Killamsetty *et al.* [16] studies the convergence of adaptive data subset selection algorithms using *stochastic gradient descent* (SGD). It is shown that the convergence bound consists of two terms: an irreducible noise-related term, and an additional gradient error term just like Eq. (16). Based on this analysis, Killamsetty *et al.* [16] propose to minimize this error directly. To this end, they use the Orthogonal Matching Pursuit (OMP) [31,7] as their greedy solver, resulting in an algorithm called GRAD-MATCH. Since GRADMATCH minimizes the gradient estimation error given in Eq. (16) objective directly, it achieves a lower error compared to CRAIG that only minimizes an upper-bound of it.

B Further Details

B.1 Final Algorithm

Alg. 1 summarizes our adversarial coreset selection approach.

Algorithm 1 Adversarial Training with Coreset Selection

Input: dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$, neural network $f_{\boldsymbol{\theta}}(\cdot)$.
Output: robustly trained neural network $f_{\boldsymbol{\theta}}(\cdot)$.
Parameters: learning rate α, total epochs E, warm-start coefficient κ, coreset update period T, batch size b, coreset size k, perturbation bound ε.

```
1: Initialize θ randomly.
2: κ_epochs = κ · E
3: T_warm = κ_epochs · k
4: for t = 1, 2, ..., E do
5:     if t ≤ T_warm then
6:         S ← D \\ Warm-start with the entire data and uniform weights.
7:     else if t ≥ κ_epochs & t%T = 0 then
8:         I = BatchAssignments(D, b) \\ Batch-wise selection.
9:         Y = {f_θ(x_i) | (x_i, y_i) ∈ D} \\ Computing the logits.
10:        G = AdvGradient(D, Y) \\ Using Eqs. 12 & 15 to find the gradients.
11:        S ← GreedySolver(D, I, G, coreset size = k) \\ Finding the coreset.
12:    else
13:        Continue
14:    end if
15:    for batch in S do
16:        batch_adv = AdvExampleGen(batch, f_θ, ε)
17:        θ ← SGD(batch_adv, f_θ, α) \\ Performing SGD over a batch of data.
18:    end for
19: end for
```

B.2 TRADES Gradient

To compute the *second* gradient term in Eq. (14) let us assume that $\boldsymbol{w}(\boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}})$ and $\boldsymbol{z}(\boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\boldsymbol{x})$. We can write the aforementioned gradient as:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}}), f_{\boldsymbol{\theta}}(\boldsymbol{x})\right) &= \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(\boldsymbol{w}(\boldsymbol{\theta}), \boldsymbol{z}(\boldsymbol{\theta})\right) \\
&\overset{(1)}{=} \frac{\partial \mathcal{L}_{\text{CE}}}{\partial \boldsymbol{w}} \nabla_{\boldsymbol{\theta}} \boldsymbol{w}(\boldsymbol{\theta}) + \frac{\partial \mathcal{L}_{\text{CE}}}{\partial \boldsymbol{z}} \nabla_{\boldsymbol{\theta}} \boldsymbol{z}(\boldsymbol{\theta}) \\
&\overset{(2)}{=} \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}}), \texttt{freeze}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x})\right)\right) \\
&\quad + \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(\texttt{freeze}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}})\right), f_{\boldsymbol{\theta}}(\boldsymbol{x})\right). \qquad (19)
\end{aligned}$$

Here, step (1) is derived using the multi-variable chain rule. Also, step (2) is the re-writing of step (1) by using the `freeze`$(\cdot)$ kernel that stops the gradients from backpropagating through its argument function. Using this derivation, we can write the final TRADES gradient as:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} \boldsymbol{\Phi} \left(\boldsymbol{x}, y; f_{\boldsymbol{\theta}}\right) = \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(f_{\boldsymbol{\theta}}(\boldsymbol{x}), y\right) &+ \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}}), \texttt{freeze}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x})\right)\right) / \lambda \\
&+ \nabla_{\boldsymbol{\theta}} \mathcal{L}_{\text{CE}} \left(\texttt{freeze}\left(f_{\boldsymbol{\theta}}(\boldsymbol{x}_{\text{adv}})\right), f_{\boldsymbol{\theta}}(\boldsymbol{x})\right) / \lambda. \qquad (20)
\end{aligned}$$

C Implementation Details

In this section, we provide the details of our experiments in Sec. 4. We used a single NVIDIA Tesla V100-SXM2-16GB GPU for CIFAR-10 [19] and SVHN [30], and a single NVIDIA Tesla V100-SXM2-32GB GPU for ImageNet-12 [32,23]. Our implementation can be found on GitHub.

C.1 Training Settings.

Tab. 5 shows the entire set of hyper-parameters and settings used for training the models of Sec. 4.

C.2 Evaluation Settings

For the evaluation of TRADES and ℓ_p-PGD adversarial training, we use PGD attacks. In particular, for TRADES and ℓ_∞-PGD adversarial training, we use ℓ_∞-PGD attacks with $\varepsilon = 8/255$, step-size $\alpha = 1/255$, 50 iterations, and 10 random restarts. Also, for ℓ_2-PGD adversarial training we use ℓ_2-PGD attacks with $\varepsilon = 80/255$, step-size $\alpha = 8/255$, 50 iterations and 10 random restarts.

For Perceptual Adversarial Training (PAT), we report the attacks' settings in Tab. 4. We evaluated each case using the same set of unseen/seen attacks as in Laidlaw *et al.* [22]. However, since we trained our model with slightly different ε bounds, we changed the attacks' settings accordingly.

Table 4: Hyper-parameters of the attacks used for the evaluation of PAT models.

Dataset	Attack	Bound	Iterations
CIFAR-10	AutoAttack-ℓ_2 [5]	1	20
	AutoAttack-ℓ_∞ [5]	8/255	20
	StAdv [43]	0.02	50
	ReColor [21]	0.06	100
	PPGD [22]	0.40	40
	LPA [22]	0.40	40
ImageNet-12	AutoAttack-ℓ_2 [5]	1200/255	20
	AutoAttack-ℓ_∞ [5]	4/255	20
	JPEG [13]	0.125	200
	StAdv [43]	0.02	50
	ReColor [21]	0.06	200
	PPGD [22]	0.35	40
	LPA [22]	0.35	40

Table 5: Training details for experimental results of Sec. 4.

Hyper-parameter	**Experiment**					
	TRADES	ℓ_∞-PGD	ℓ_2-PGD	Fast-LPA	Fast-LPA	Fast Adv.
Dataset	CIFAR-10	CIFAR-10	SVHN	CIFAR-10	ImageNet-12	CIFAR-10
Model Arch.	ResNet-18	ResNet-18	ResNet-18	ResNet-50	ResNet50	ResNet-18
Optimizer	SGD	SGD	SGD	SGD	SGD	SGD
Scheduler	Multi-step	Multi-step	Multi-step	Multi-step	Multi-step	Multi-step
Initial lr.	0.1	0.01	0.1	0.1	0.1	0.1
lr. Decay (epochs)	0.1 (75, 90)	0.1 (80, 100)	0.1 (75, 90, 100)	0.1 (75, 90, 100)	0.1 (45, 60, 80))	0.1 (37, 56)
Weight Decay	$2 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$5 \cdot 10^{-4}$	$2 \cdot 10^{-4}$	$2 \cdot 10^{-4}$	$5 \cdot 10^{-4}$
Batch Size (full)	128	128	128	50	50	128
Total Epochs	100	120	120	120	90	60
Coreset Size	50%	50%	30%	40%	50%	50%
Coreset Batch Size	20	20	20	20	20	20
Warm-start Epochs	30	36	22	29	27	22
Coreset Selection Period (epochs)	20	20	20	10	15	5
Visual Similarity Measure	ℓ_∞	ℓ_∞	ℓ_2	LPIPS (AlexNet [20])	LPIPS (AlexNet)	ℓ_∞
ε **(Bound on Visual Sim.)**	8/255	8/255	80/255	0.5	0.25	8/255
Attack Iterations (Training)	10	10	10	10	10	1
Attack Iterations (Coreset Selection)	10	1	10	10	10	1
Attack Step-size	1.785/255	1.25/255	8/255	-	-	10/255

D Extended Experimental Results

D.1 Extended Results of PAT vs. Unseen Attacks

Tab. 6 shows the full details of our experiments on PAT [22]. In each case, we train ResNet-50 [12] classifiers using LPIPS [45] objective of PAT [22]. All the training hyper-parameters are fixed. The only difference is that we enable adversarial coreset selection as our method. During inference, we evaluate each trained model against a few unseen attacks, as well as two variants of Perceptual Adversarial Attacks [22] that the models are trained initially on. As can be seen, adversarial coreset selection can significantly reduce the training time while experiencing only a tiny reduction in the average robust accuracy.

D.2 Trade-offs

Here, we study the accuracy vs. speed-up trade-off in adversarial coreset selection. For this study, we train our adversarial coreset selection method using different versions of Craig [27] and GradMatch [16] on CIFAR-10 using TRADES. In particular, for each method, we start with the base algorithm and add the batch-wise selection and warm-start one by one. Also, to capture the effect of the coreset size, we vary this number from 50% to 10% in each case. Fig. 4 shows the clean and robust error vs. speed-up compared to full adversarial training. In each case, the combination of warm-start and batch-wise versions of the adversarial coreset selection gives the best performance. Moreover, the training speed increases as we gradually decrease the coreset size. However, this gain in training speed is achieved at the cost of increasing the clean and robust error. Both of these observations are in line with that of Killamsetty *et al.* [16] around vanilla coreset selection.

D.3 Training with a Mixture of Coreset and Non-coreset Data

In this section, we run an experiment similar to that of Tsipras *et al.* [37]. Specifically, we minimize the average of adversarial and vanilla training in each epoch. The non-coreset data is treated as clean samples to minimize the vanilla objective, while for the coreset samples, we would perform adversarial training. Tab. 7 shows the results of this experiment. As seen, adding the non-coreset data as clean inputs to the training improves the clean accuracy while decreasing the robust accuracy. These results align with the observations of Tsipras *et al.* [37] around the existence of a trade-off between clean and robust accuracy.

Table 6: Clean and robust accuracy (%) and total training time (mins) of Perceptual Adversarial Training for CIFAR-10 and ImageNet-12 datasets. The training objective uses Fast Lagrangian Perceptual Attack (LPA) [22] to train the network. At test time, the networks are evaluated against attacks not seen during training and different versions of Perceptual Adversarial Attack (PPGD and LPA). In each dataset, the unseen attacks were selected similar to Laidlaw *et al.* [22]. Please see the Appendix C for more information about the settings.

Dataset	Training Method	Clean	Unseen Attacks						Seen Attacks		Train. Time (mins)
			Auto-ℓ_2 [5]	Auto-ℓ_∞ [5]	JPEG [13]	StAdv [43]	ReColor[21]	Mean	PPGD	LPA	
CIFAR-10	Adv. Craig (Ours)	**83.21**	39.98	33.94	-	49.60	62.69	**46.55**	19.56	7.42	**767.34**
	Adv. GradMatch (Ours)	**83.14**	39.20	34.11	-	48.86	62.26	**46.11**	19.94	7.54	**787.26**
	Full PAT (Fast-LPA)	**86.02**	43.27	37.96	-	48.68	62.23	**48.04**	22.62	8.01	**1682.94**
ImageNet-12	Adv. Craig (Ours)	**86.99**	51.54	60.42	71.79	37.47	44.04	**53.05**	29.04	14.07	**2817.06**
	Adv. GradMatch (Ours)	**87.08**	51.38	60.64	72.15	35.83	45.83	**53.17**	28.36	13.11	**2865.72**
	Full PAT (Fast-LPA)	**91.22**	57.37	66.89	76.25	19.29	46.35	**53.23**	33.17	13.49	**5613.12**

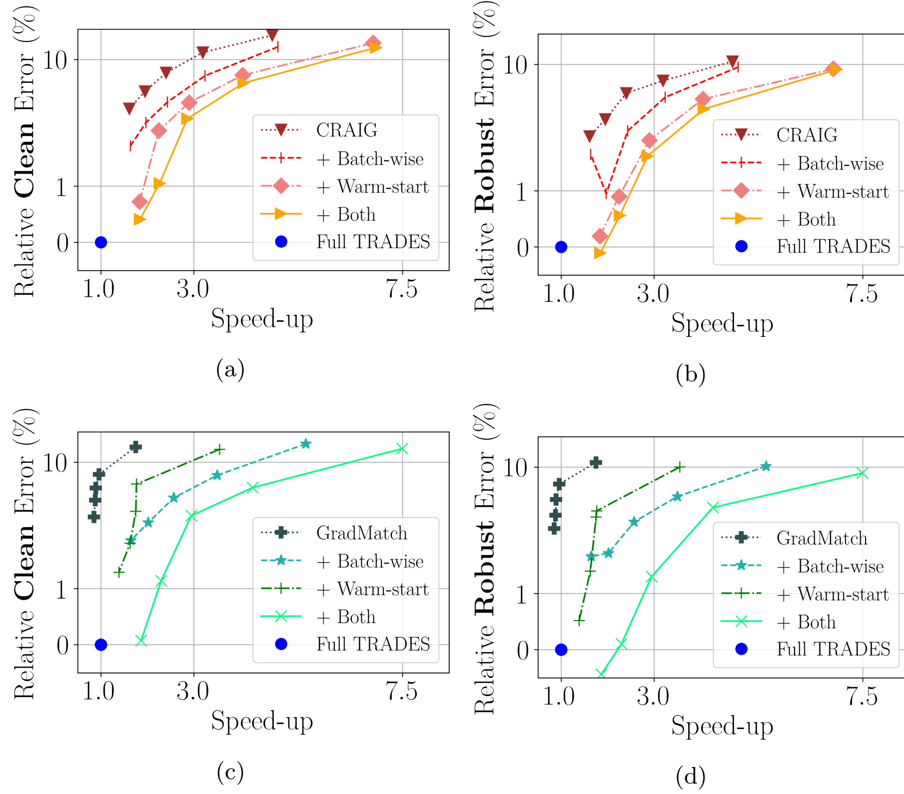

Fig. 4: Relative error vs. speed up curves for different versions of adversarial coreset selection in training CIFAR-10 models using the TRADES objective. In each figure, the coreset size is changed from 50% to 10% (left to right). (a, b) Clean and robust error vs. speed up compared to full TRADES for different versions of adversarial CRAIG. (c, d) Clean and robust error vs. speed up compared to full TRADES for different versions of adversarial GRADMATCH.

Table 7: Performance of ℓ_∞-PGD over CIFAR-10. In "Half-Half", we mix half adversarial coreset selection samples with another half of clean samples and train a neural network similar to [37]. In "ONLY-Core" we just use adversarial coreset samples. Settings are given in Tab. 5. The results are averaged over 5 runs.

Training Method	↑ **Clean** (%)		↑ **RACC** (%)		↓ **T** (mins)	
	ONLY Core	**Half-Half**	**ONLY Core**	**Half-Half**	**ONLY Core**	**Half-Half**
Adv. Craig	80.36	84.43	45.07	39.83	148.01	152.34
Adv. GradMatch	80.67	84.31	45.23	40.05	148.03	153.18
Full Adv. Training	83.14		41.39		292.87	