A. Proofs

In the proofs, we use abbreviated notation by dropping x and y and making θ a subscript, e.g., we write f_θ for $f(\theta; x)$.

A.1. Proof of Proposition 2.1

Proposition 2.1 *When $h(p) = L_y(p)$ that returns loss given prediction p, Algorithm 1 with $\alpha = \gamma$ is equivalent to Algorithm 3 with $\alpha = 1 - (1 - \gamma)^m$.*

Proof From Algorithm 1 with $\alpha = \gamma$, we have

$$f_i^* = \arg\min_q \left[D_h(q, f_{i-1}^*) + \gamma \nabla L_y(f_{i-1}^*)^\top q \right]. \tag{15}$$

From $h(\cdot) = L_y(\cdot)$ and (15), we obtain

$$\nabla L_y(f_i^*) = \nabla L_y(f_{i-1}^*) - \gamma \nabla L_y(f_{i-1}^*) = (1 - \gamma) \nabla L_y(f_{i-1}^*) \quad \text{for } i = 1, \cdots, m.$$

Since $f_0^* = f_{\theta_t}$, we have

$$\nabla L_y(f_m^*) = (1 - \gamma)^m \nabla L_y(f_0^*) = (1 - \gamma)^m \nabla L_y(f_{\theta_t}),$$

which implies

$$\begin{aligned} \nabla_{f_\theta} \left[D_h(f_\theta, f_m^*) \right] &= \nabla L_y(f_\theta) - \nabla L_y(f_m^*) = \nabla L_y(f_\theta) - (1 - \gamma)^m \nabla L_y(f_{\theta_t}) \\ &= \nabla_{f_\theta} \left[D_{L_y}(f_\theta, f_{\theta_t}) + (1 - (1 - \gamma)^m) \nabla L_y(f_{\theta_t})^\top f_\theta \right] \end{aligned}$$

and therefore, $\nabla_\theta \left[D_h(f_\theta, f_m^*) \right] = \nabla_\theta \left[D_{L_y}(f_\theta, f_{\theta_t}) + (1 - (1 - \gamma)^m) \nabla L_y(f_{\theta_t})^\top f_\theta \right]$. The rest is trivial. ■

A.2. Proof of Proposition 2.2

Proposition 2.2 *Let y be a vector representation such as a K-dim vector representing K classes. Assume that the gradient of the loss function can be expressed as*

$$\nabla L(f, y) = \nabla L_y(f) = p(f) - y$$

with $p(f)$ not depending on y. Let

$$\begin{aligned} J_t(\theta) &= \left\langle D_{L_y}(f_\theta, f_{\theta_t}) + \alpha \nabla L_y(f_{\theta_t})^\top f_\theta \right\rangle_{(x,y) \in S} \\ J_t'(\theta) &= \left\langle (1 - \alpha) L(f_\theta, p(f_{\theta_t})) + \alpha L_y(f_\theta) \right\rangle_{(x,y) \in S} \end{aligned}$$

Then we have

$$J_t(\theta) = J_t'(\theta) + c_t,$$

where c_t is independent of θ. This implies that

$$\arg\min_\theta \left[J_t(\theta) + R(\theta) \right] = \arg\min_\theta \left[J_t'(\theta) + R(\theta) \right].$$

Proof

$$\begin{aligned} \nabla_{f_\theta} \left[D_{L_y}(f_\theta, f_{\theta_t}) + \alpha \nabla L_y(f_{\theta_t})^\top f_\theta \right] &= \nabla L_y(f_\theta) - (1 - \alpha) \nabla L_y(f_{\theta_t}) \\ &= (p(f_\theta) - y) - (1 - \alpha)(p(f_{\theta_t}) - y) \\ &= (1 - \alpha)(p(f_\theta) - p(f_{\theta_t})) + \alpha(p(f_\theta) - y) \\ &= \nabla_{f_\theta} \left[(1 - \alpha) L(f_\theta, p(f_{\theta_t})) + \alpha L_y(f_\theta) \right]. \end{aligned}$$

This implies that $\nabla J_t(\theta) = \nabla J_t'(\theta)$. Therefore $J_t(\theta) - J_t'(\theta)$ is independent of θ. ■

A.3. Proof of Theorem 2.1

Theorem 2.1 *In the setting of Algorithm 1 with $m = 1$, assume that there exists $\beta > 0$ such that $D_h(f, f') \geq \beta D_{L_y}(f, f')$ for any f and f', and assume that $\alpha \in (0, \beta]$. Assume also that $Q_t(\theta)$ defined in Algorithm 1 is $1/\eta$ smooth in θ:*

$$\|\nabla Q_t(\theta) - \nabla Q_t(\theta')\| \leq (1/\eta)\|\theta - \theta'\|.$$

Assume that θ_{t+1} is an improvement of θ_t with respect to minimizing Q_t so that

$$Q_t(\theta_{t+1}) \leq Q_t(\tilde{\theta}),$$

where

$$\tilde{\theta} = \theta_t - \eta \nabla Q_t(\theta_t).$$

Then we have

$$\ell_\alpha(\theta_{t+1}) \leq \ell_\alpha(\theta_t) - \frac{\alpha\eta}{2}\|\nabla \ell_\alpha(\theta_t)\|^2.$$

Proof

We first define $\tilde{Q}_t(\theta)$ as follows:

$$\tilde{Q}_t(\theta) := \left\langle D_h(f_\theta, f_{\theta_t}) + \alpha \nabla L_y(f_{\theta_t})^\top f_\theta \right\rangle_{(x,y)\in S} + R(\theta).$$

We can check that $Q_t(\theta) - \tilde{Q}_t(\theta)$ is independent of θ. Therefore optimizing θ with respect to $Q_t(\theta)$ is the same as optimizing θ with respect to $\tilde{Q}_t(\theta)$, and $\nabla Q_t(\theta) = \nabla \tilde{Q}_t(\theta)$.

The smoothness assumption implies that

$$\tilde{Q}_t(\theta - \Delta\theta) \leq \tilde{Q}_t(\theta) - \nabla Q_t(\theta)^\top \Delta\theta + \frac{1}{2\eta}\|\Delta\theta\|^2.$$

Therefore

$$\begin{aligned}
\tilde{Q}_t(\theta_{t+1}) \leq & \tilde{Q}_t(\tilde{\theta}) = \tilde{Q}_t(\theta_t - \eta\nabla Q_t(\theta_t)) \\
\leq & \tilde{Q}_t(\theta_t) - \eta\|\nabla Q_t(\theta_t)\|^2 + \frac{1}{2\eta}\|\eta\nabla Q_t(\theta_t)\|^2 \\
= & \tilde{Q}_t(\theta_t) - \frac{\eta}{2}\|\nabla Q_t(\theta_t)\|^2.
\end{aligned}$$

Note also that

$$\begin{aligned}
\tilde{Q}_t(\theta_{t+1}) - \tilde{Q}_t(\theta_t) \geq & \left\langle \beta D_{L_y}(f_{\theta_{t+1}}, f_{\theta_t}) + \alpha\nabla L_y(f_{\theta_t})^\top (f_{\theta_{t+1}} - f_{\theta_t}) \right\rangle_{(x,y)\in S} + [R(\theta_{t+1}) - R(\theta_t)] \\
= & \left\langle (\beta - \alpha) D_{L_y}(f_{\theta_{t+1}}, f_{\theta_t}) + \alpha L_y(f_{\theta_{t+1}}) - \alpha L_y(f_{\theta_t}) \right\rangle_{(x,y)\in S} + [R(\theta_{t+1}) - R(\theta_t)] \\
\geq & \left\langle \alpha L_y(f_{\theta_{t+1}}) - \alpha L_y(f_{\theta_t}) \right\rangle_{(x,y)\in S} + [R(\theta_{t+1}) - R(\theta_t)] \\
= & \alpha\ell_\alpha(\theta_{t+1}) - \alpha\ell_\alpha(\theta_t).
\end{aligned}$$

The second inequality is due to the non-negativity of the Bregman divergence.

By combining the two inequalities, we obtain

$$\alpha\ell_\alpha(\theta_{t+1}) \leq \alpha\ell_\alpha(\theta_t) - \frac{\eta}{2}\|\nabla Q_t(\theta_t)\|^2.$$

Now, observe that $\nabla Q_t(\theta_t) = \nabla \tilde{Q}_t(\theta_t) = \alpha \nabla \ell_\alpha(\theta_t)$, and we obtain the desired bound. ■

B. On the Empirical Study

In this section, we first provide experimental details and additional figures regarding the experiments reported in the main paper, and then we report additional experiments using text data. Our code is provided at a repository under `github.com/riejohnson`.

B.1. Details of the experiments in the main paper

B.1.1. CIFAR10, CIFAR100, AND SVHN

This section describes the experimental details of all but the ImageNet experiments.

The mini-batch size was set to 128. We used momentum 0.9. The following learning rate scheduling was used: 200K steps with η, 40K steps with 0.1η, and 40K steps with 0.01η. The initial learning rate η was set to 0.1 on CIFAR10/100 and 0.01 on SVHN, following (Zagoruyko & Komodakis, 2016). The weight decay λ was 0.0001 except that it was 0.0005 for (CIFAR100, WRN-28-10) and SVHN.

We used the standard mean/std normalization on all and the standard shift and horizontal flip image augmentation on CIFAR10/100.

We report the median of three runs with three random seeds. The meta-parameters were chosen based on the performance on the development set. All the results were obtained by using only the 'train' portion (shown in Table 1 of the main paper) of the official training set as training data.

For label smoothing, the amount of probability taken away from the true class was chosen from $\{0.1, 0.2, 0.3, 0.4\}$.

To obtain the results reported in Table 2 (with smaller networks), T was fixed to 25 for CIFAR10/100, and 15 for SVHN. α for ini:random was fixed to 0.3. For ini:base, we chose α from $\{0.3, 0.01\}$. We excluded $\alpha = 0.01$ for ini:random, as it takes too long. When dropout was applied in the SVHN experiments, the dropout rate was set to 0.4, following (Zagoruyko & Komodakis, 2016). To obtain the results reported in Table 3 (with larger networks), T was fixed to 1. For GULF2, α was chosen from $\{0.3, 0.01\}$. For GULF1, α was fixed to 0.3, and m (the number of functional gradient steps) was chosen from $\{1, 2, 5\}$. On CIFAR datasets, the choice of α or m did not make much difference, and the chosen values tended to vary among the random seeds. On SVHN, α=0.01 tended to be better when no dropout was used, and 0.3 was better when dropout was used.

To perform random initialization of the parameter for ini:random and the baseline methods, we used Kaiming normal initialization (He et al., 2015), following the previous work.

B.1.2. IMAGENET

Each stage of the training for ImageNet followed the code used for training the pre-trained models provided as part of TorchVision: `https://github.com/pytorch/examples/blob/master/imagenet/main.py`. That is, for both ResNet-50 and WRN-50-2, the learning rate was set to η, 0.1η, and 0.01η for 30 epochs each, i.e., 90 epochs in total, and the initial rate η was set to 0.1. The mini-batch size was set to 256, and the weight decay was set to 0.0001. The momentum was 0.9. α was fixed to 0.5. We used two GPUs for ResNet-50 and four GPUs for WRN-50-2.

We used the standard mean/std normalization and the standard image augmentation for ImageNet – random resizing, cropping and horizontal flip, which is the same data augmentation scheme as used for training the pre-trained models provided as part of TorchVision.

B.2. Additional figures

Figure 5 shows test error (%) in relation to training loss with a small ResNet on CIFAR100. Additional examples of test-loss curves are shown in Figure 6. Figure 7 shows the parameter size $\|\theta_t\|^2$ in relation to training loss, in the settings of Figure 2 in the main paper.

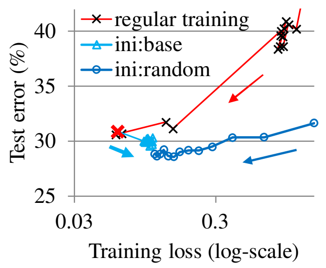
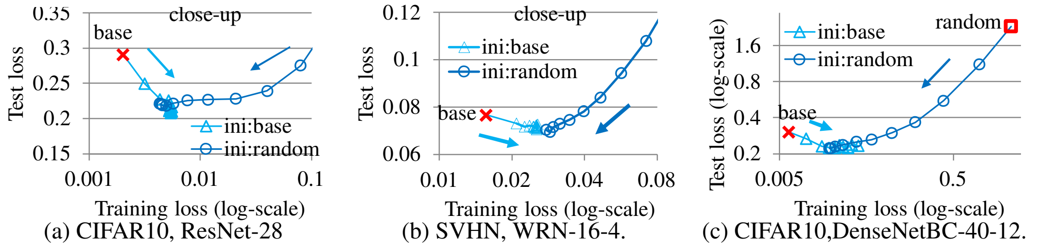
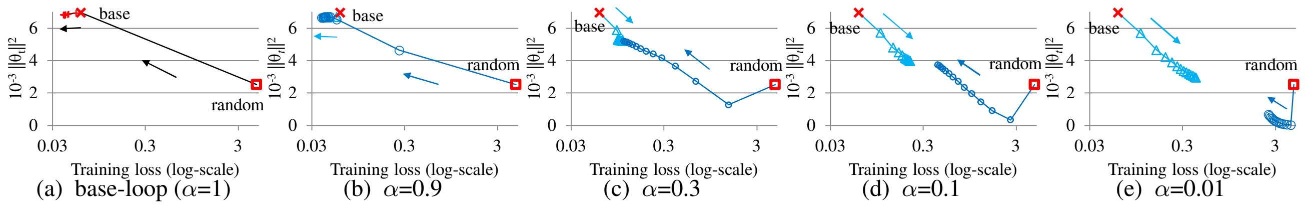


Figure 5. Test error (%) in relation to training loss. The arrows indicate the direction of time flow. GULF2. CIFAR100. ResNet-28.



(a) CIFAR10, ResNet-28 (b) SVHN, WRN-16-4. (c) CIFAR10,DenseNetBC-40-12.

Figure 6. Additional examples of test loss curves of GULF2. The arrows indicate the direction of time flow.



(a) base-loop (α=1) (b) α=0.9 (c) α=0.3 (d) α=0.1 (e) α=0.01

Figure 7. Parameter size $\|\theta_t\|^2$ of ini:base('△') and ini:random('∘'). with five values of α (becoming smaller from left to right), in relation to training loss. GULF2. T=25. CIFAR100. ResNet-28. Matching figures with Figure 2. As α becomes smaller, the (potential) meeting point shifts further away from the base model. The left-most figure is base-loop, which is equivalent to α=1. The arrows indicate the direction of time flow.

B.3. Additional experiments on text data

We tested GULF on sentiment classification to predict whether reviews are positive or negative, using the polarized Yelp dataset (#train: 560K, #test: 38K) (Zhang et al., 2015). The best-performing models on this task are transformers pre-trained with language modeling on large and general text data such as Bert (Devlin et al., 2019) and XLnet (Yang et al., 2019b). However, these models are generally large and time-consuming to train using a GPU (i.e., without TPUs used in the original work). Therefore, instead, we used the deep pyramid convolutional neural network (DPCNN) (Johnson & Zhang, 2017) as our base model. In these experiments, we used GULF2.

Table 5 shows the test error results in five settings. The last three use relatively small training sets of 45K data points and validation sets of 5K data points, randomly chosen from the original training set, while the first two use the entire training set (560K data points) except for 5K data points held out for validation (meta-parameter tuning). DPCNNs optionally take additional features produced by embeddings of text regions that are trained with unlabeled data, similar to language modeling. Cases 1 and 3 exploited this option, training embeddings using the entire training set as unlabeled data; B.3.1 below provides the details. As in the image experiments, we used the cross entropy loss with softmax except for Case 5, where the quadratic hinge loss $L_y(f) = \max(0, 1-yf)^2$ for $y \in \{-1, 1\}$ was used. This serves as an example of extending

Case#		1	2	3	4	5
Data		large-Yelp		small-Yelp		
Embedding learning?		Yes	No	Yes	No	
Loss function		cross-entropy				†
baselines	base model	2.81	2.98	3.80	5.43	5.32
	base-loop	2.63	2.88	3.90	5.43	5.34
	w/ dropout	2.70	2.95	3.90	5.34	5.35
GULF	ini:random	**2.34**	2.72	**3.70**	5.06	5.00
	ini:base	2.38	**2.70**	3.77	5.15	4.98
	ini:base/2	2.43	2.74	3.73	**4.99**	**4.96**

Table 5. Test error (%) on sentiment classification. Median of 3 runs. 7-block 250-dim DPCNN (10M parameters). † Squared hinge loss.

self-distillation (formulated specifically with the cross-entropy loss) to general loss functions.

In all the five settings, GULF achieves better test errors than the baseline methods, which shows the effectiveness of our approach in these settings. On this task, dropout turned out to be not very effective, which is, however, a reminder that the effectiveness of regularization methods can be data-dependent in general.

Case#		1	2		
		LM-like prep?		Runtime	Text for
		Yes	No	(sec/K)	prep (GB)
(J & Z, 2017)	DPCNN	*2.64*	*3.30*	0.1	0.4
This work	Table 5 best	2.34	2.70	0.1	0.4
	Ensemble	**2.18**	**2.46**	0.9	0.4
(Devlin et al., 2019)	Bert base	2.25	6.19	5.7	13
	Bert large	*1.89*	–	17.9	13
(Yang et al., 2019b)	XLnet base	1.92	4.51	17.2	13
	XLnet large	*1.55*	–	40.5	126

Table 6. GULF ensemble results on Yelp in comparison with previous models. Test error (%) with or without embedding learning (DPCNN) or language modeling-based pre-training (Bert and XLnet), respectively, corresponding to Cases 1 & 2 of Table 5. Runtime: real time in seconds for labeling 1K instances using a single GPU with 11GB device memory, measured in the setting of Case 1; the average of 3 runs. The last column shows amounts of text data in giga bytes used for pre-training or embedding learning in Case 1. The test errors in *italics* were copied from the respective publications except that the Bert-large test error is from (Xie et al., 2019); other test errors and runtime were obtained by our experiments. Our ensemble test error results are in bold.

It is known that performance can be improved by making an ensemble of models from different stages of self-distillation, e.g., (Furlanello et al., 2018). In Table 6, we report ensemble performances of DPCNNs trained with GULF, in comparison with the previous best models. Test errors with and without embedding learning (or language modeling-based pre-training for Bert and XLnet) are shown, corresponding to Cases 1 and 2 in Table 5. The ensemble results were obtained by adding after applying softmax the output values of 20 DPCNNs (or 10 in Case 2) of last 5 stages of GULF training with different training options; details are provided in B.3.1.

With embedding learning, the ensemble of DPCNNs trained with GULF achieved test error 2.18%, which slightly beats 2.25% of pre-trained Bert-base, while testing (i.e., making predictions) of this ensemble is more than 6 times faster than Bert-base, as shown in the 'Runtime' column. (Note, however, that runtime depends on implementation and hardware/software configurations.) That is, using GULF, we were able to obtain a classifier that is as accurate as and much faster than a pre-trained transformer.

(Yang et al., 2019b) and (Xie et al., 2019) report 1.55% and 1.89% using a pre-trained large transformer, XLnet-large and Bert-large, respectively. We observe that the runtime and the amounts of text used for pre-training (the last two columns) indicate that their high accuracies come with steep cost at every step: pre-training, fine-tuning, and testing. Compared with them, an ensemble of GULF-trained DPCNNs is a much lighter-weight solution with an appreciable accuracy. Also, our ensemble without embedding learning outperforms Bert-base and XLnet-base without pre-training, with relatively large differences (Case 2). A few attempts of training Bert-large and XLnet-large from scratch also resulted in underperforming DPCNNs, but we omit the results as we found it infeasible to complete meta-parameter tuning in reasonable time.

On the other hand, it is plausible that the accuracy of the high-performance pre-trained transformers can be further improved by applying GULF to their fine-tuning, which would further push the state of the art. Though currently precluded by our computational constraints, this may be worth investigating in the future.

B.3.1. Details of the text experiments

Embedding learning It was shown in (Johnson & Zhang, 2017) that classification accuracy can be improved by training an embedding of small text regions (e.g., 3 consecutive words) for predicting neighboring text regions ('target regions') on unlabeled data (similar to language modeling) and then using the learned embedding function to produce additional features for the classifier. In this work, we trained the following two types of models with respect to use of embedding learning.

- Type-0 did not use any additional features from embedding learning.
- Type-1 used additional features from the following two types of embedding simultaneously:
  - the embedding of 3-word regions as a function of a bag of words to a 250-dim vector, and
  - the embedding of 5-word regions as a function of a bag of word $\{1,2,3\}$-grams to a 250-dim vector.

Embedding training was done using the entire training set (560K reviews, 391MB) as unlabeled data disregarding the labels.

It is worth mentioning that our implementation of embedding learning differs from the original DPCNN work (Johnson & Zhang, 2017), as a result of pursuing an efficient implementation in pyTorch (the original implementation was in C++). The original work used the bag-of-word representation for target regions (to be predicted) and minimized squared error with negative sampling. In this work we minimized the log loss without sampling where the target probability was set by equally distributing the probability mass among the words in the target regions.

Table 5 Optimization was done by SGD. The learning rate scheduling of the base model and each stage of base-loop and GULF was fixed to 9 epochs with the initial learning rate η followed by 1 epoch with 0.1η. The mini-batch size was 32 for small training data and 128 for large training data. We chose the weight decay parameter from $\{$1e-4, 2e-4, 5e-4, 1e-3$\}$ and the initial learning rate from $\{0.25, 0.1, 0.05\}$, using the validation data, except that for GULF on the large training data, we simply used the values chosen for the base model, which were weight decay 1e-4 and learning rate 0.1 (with embedding learning) and 0.25 (without embedding learning).

For GULF, we chose the number of stages T from $\{1,2,\ldots,25\}$ and α from $\{0.3, 0.5\}$, using the validation data. $\alpha = 0.5$ was chosen in most cases.

Table 6 The ensemble performances were obtained by combining

- 20 DPCNNs ($T \in \{21, 22, \ldots, 25\} \times \{\text{ini:random, ini:base}\} \times \{\text{Type-0, Type-1}\}$) in Case 1, and
- 10 DPCNNs ($T \in \{21, 22, \ldots, 25\} \times \{\text{ini:random, ini:base}\} \times \{\text{Type-0}\}$) in Case 2.

To make an ensemble, the model output values were added after softmax.

Transformers The Bert and XLnet experiments were done using HuggingFace's Transformers[2] in pyTorch. Following the original work, optimization was done by Adam with linear decay of learning rate. For enabling and speeding up training using a GPU, we combined the techniques of gradient accumulation and variable-sized mini-batches (for improving parallelization) so that weights were updated after obtaining the gradients from approximately 128 data points. 128 was chosen, following the original work. To measure runtime of transformer testing, we used variable-sized mini-batches for speed-up by improving the parallelism on a GPU.

[2] `https://huggingface.co/transformers/`