A Experimental Details

In this section, we show the details of the experimental setting used in Section 4.

In this work, the BERT model, which is called *bert-base-cased* in the Transformers library of Wolf et al. (2019), was used. The hyperparameters were selected based on the validation accuracy in preliminary experiments. These hypterparameters were: learning rate = 5e-5 (from [2e-5, 5e-5]), batch size = 32, optimization epoch = 3 (from [3, 6, 10]), which were chosen by grid search based on the validation accuracy of SST-2. For the models using the turn-over dropout, 10 epochs are used; while even training of 3 epochs worked and achieved the same accuracy, 10 epochs were a little more stable for estimation.

For the learning curves in Figure 4.2, the model is trained with 25,000 training instances for presenting the curves with less noise.

For the dataset of Multi-Domain Sentiment Dataset (Blitzer et al., 2007) can be downloaded from `http://www.cs.jhu.edu/~mdredze/datasets/sentiment/`. We extracted the 'electronics' subset from their unprocessed dataset, and tokenized the texts using Stanza (Qi et al., 2020) to align the input format as the SST-2 is already tokenized.

VGGNet19 was trained with the momentum SGD method with momentum = 0.9 and weight decay = 5e-4. Moreover, a decaying learning rate by 0.1 was applied at the 150th and 225th epochs from the initial rate = 0.1, without the data augmentation of horizontal flip. The implementation was derived from `https://github.com/kuangliu/pytorch-cifar`.

B Runtime Compared with Koh and Liang (2017)

Han et al. (2020) reported that Koh and Liang (2017)'s method for BERT on the Multi-Genre NLI dataset (Williams et al., 2018) took 10 minutes to estimate the influences of 10,000 training instances on another single instance, using one NVIDIA GeForce RTX 2080 Ti GPU. In our experiment, our proposed method took 35 seconds (i.e., 17 times faster) to estimate the same influences from the same dataset on the same GPU in our environment too. While some accidental implementations may differ, both implementations of BERT are derived from Wolf et al. (2019)'s one.

In addition to the efficiency indicated by the big-O notation in Table 1, our method can also process different training instances in a mini-batch efficiently at once because it only uses forward computations, unlike the others. This is another advantage of our method in terms of efficiency.

C Hash-based Mask Composition

The volatile mask generation method (Section 3.3) solved storage and memory issues in our experiments. However, memory issues (i.e., high space complexity) could occur even with the method, depending on implementations and dataset sizes.

For example, as a typical efficient implementation, mask generation for all instances in a mini-batch should be performed with a few operations. We can implement it with the two operations; at each layer during the forward computation, we (1) generate all instance-specific masks from a random seed and (2) extract a subset, which is required for the current mini-batch, by indexing. In this case, the first step temporally requires large memory space, which depends on the dataset size and the layer's dimension, while it volatilizes after the second step. As a solution for mitigating memory usage, we can use *hash-based mask composition*. The basic idea is that we can generate different N random masks from combinations of K ($\ll N$) random masks.

We first generate a codebook composed of K binary random masks, each of which is a d-dimensional dropout mask sampled from Bernoulli$(1 - p^{1/k})$. We also prepare a hash function $H\colon Z \to \{1, ..., K\}^k$, which deterministically converts an instance $z \in Z$ to k integers so that we can pick k rows of the codebook. Using the two components, given an instance, we can deterministically obtain k primitive dropout masks, whose elements' ratio of 1 is $1 - p^{1/k}$. Finally, performing cumulative product (or logical-AND) of the k masks, we obtain a binary mask whose ratio of 1 is p in expectation. After scaling it with the factor $\frac{1}{p}$, we can use it as a dropout mask. This procedure can be implemented as fast batch processing using typical array operations only. We can share the codebook with different layers in a network if using different hash functions so that different layers use different dropout masks. Therefore, the space complexity of this algorithm is only $O(K\delta)$, where δ is the maximum dimension of a layer in a network. And also, since the codebook can be used

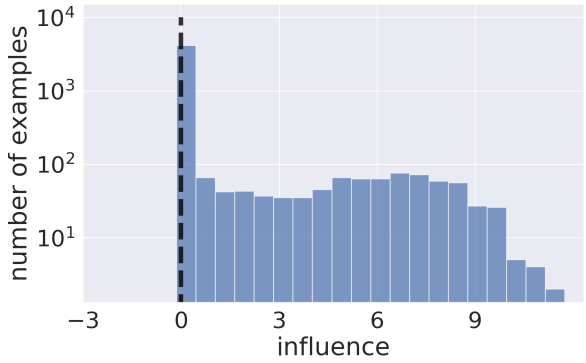

Figure 6: Histogram of self-influence of BERT on SST-2.

based on the volatile mask generation, it can avoid saving this codebook, i.e., reducing the required storage size to a minimum.

In summary, volatile mask generation reduces storage space and long-term memory space, and hash-based mask composition reduces both long and short-term memory spaces, while both require some computations for generation or composition. The two techniques make instance-specific parameters applicable to large datasets.

D Self-influence of Training Instances

We have conducted a preliminary experiment to analyze the estimated influences. A common belief in supervised learning is that the model should achieve lower loss on the training instances. For validating it, for BERT on SST-2, we estimated the influence of each training instance on a prediction of the instance itself (*self-influence*) and presented histograms in Figure 6. VGGNet also showed a similar distribution. We can see that most of the instances have positive (> 0) influence on themselves. The results agree with the hypothesis that most of the training instances have positive influences on themselves.