Estimating Training Data Influence by Tracking Gradient Descent

Garima *1 Frederick Liu *1 Satyen Kale 1 Mukund Sundararajan

Abstract

We introduce a method called TrackIn that computes the influence of a training example on a prediction made by the model, by tracking how the loss on the test point changes during the training process whenever the training example of interest was utilized. We provide a scalable implementation of TrackIn via a combination of a few key ideas: (a) a first-order approximation to the exact computation, (b) using random projections to speed up the computation of the first-order approximation for large models, (c) using saved checkpoints of standard training procedures, and (d) cherry-picking layers of a deep neural network. An experimental evaluation shows that TrackIn is more effective in identifying mislabelled training examples than other related methods such as influence functions and representer points. We also discuss insights from applying the method on vision, regression and natural language tasks.

1. Motivation

Deep learning has been used to solve a variety of real-world problems. For instance, the prediction of disease from medical imaging, recommending and ranking content, question-answering over databases and text corpora, etc. A common form of machine learning is supervised learning, where the model is trained on labelled data. For instance, there are models that are used to predict diabetic retinopathy, a degenerative eye condition, from fundus images of the retina (cf. (Gulshan et al., 2016)). These models are trained on images labelled by ophthalmologists. Labelling processes are prone to error. Furthermore, they can also be somewhat subjective. For diabetic retinopathy, while there are some widely accepted diagnosis guidelines, there is also disagreement across doctors (see for instance eFigure3 in the supplement of (Gulshan et al., 2016)). Controlling the training data input to the model is one of the main quality knobs to improve the quality of the deep learning model. For instance, such a technique could be used to identify and fix mislabelled data using the workflow described in Section 4.2.

Our main motivation is to identify practical techniques to improve the analysis of the training data. Specifically, we study the problem of identifying the influence of training examples on the prediction of a test example.

2. Related Work

(Bien & Tibshirani, 2011; Kim et al., 2014) approach the problem as the problem of identifying small set of prototypical training points, and use those to perform prediction. (Bien & Tibshirani, 2011) uses a set-cover algorithm to pick a few points that cover the rest. (Kim et al., 2014) adopts a clustering approach to learn prototypes during the training process. (Kim et al., 2016) additionally emphasizes criticisms, i.e., examples that are not adequately predicted by the prototypes. These works don’t study deep learning.

(Koh & Liang, 2017; Yeh et al., 2018) tackle influential training examples in the context of deep learning. (Koh & Liang, 2017), uses a classic technique from robust statistics called influence functions. Influence functions mimic the process of tracking the change in an individual prediction when you drop an individual training point and retrain. Implementing this directly would be prohibitively expensive. Therefore, the influence functions approach approximates this by using the first and second order optimality conditions. Unfortunately this involves inversion of a Hessian matrix that has a size that is quadratic in the number of model parameters, making the approach costly. (Yeh et al., 2018) computes the influence of training point using the representer theorem, which posits that when only the top layer of a neural network is trained with \( \ell_2 \) regularization, the obtained model parameters can be specified as a linear combination of the post-activation values of the training points at the last layer.

There are related notions of influence used to explain deep learning models that differ in either the target of the explanation or the choice of influencer or both. For instance, (Sundararajan et al., 2017; Lundberg & Lee, 2017; Ribeiro et al., 2016) identify the influence of features on an individ-
We measure the performance of a predictor via a loss function. We will then develop practical approximations that resolve a one-shot explanation and the representer point approach) are asymmetric. Techniques that compute an example similarity measure could also be used instead of influence methods. (One commonly used model-based similarity measure is the distance between activation vectors of the two examples.) The premise is that a training example that is influential for a specific test example’s prediction is also likely to be similar to it. However, it is also possible that influential examples do not resemble the test example, but nevertheless affect its prediction via certain model parameters. Furthermore, similarity is a symmetric concept, whereas all influence measures (including TrackIn, the influence functions approach and the representer point approach) are asymmetric. Thus similarity and influence are conceptually different.

3. The Method

In this section we define TrackIn. We start with an idealized definition to clarify the idea, but this definition will be impractical because it would require that the test examples (the ones to be explained) to be specified at training time. We will then develop practical approximations that resolve this constraint.

3.1. Idealized Notion of Influence

Let \( Z \) represent the space of examples, and we represent training or test examples in \( Z \) by the notation \( z, z' \) etc. We train predictors parameterized by a weight vector \( w \in \mathbb{R}^p \).

We measure the performance of a predictor via a loss function \( \ell : \mathbb{R}^p \times Z \rightarrow \mathbb{R} \); thus, the loss of a predictor parameterized by \( w \) on an example \( z \) is given by \( \ell(w, z) \).

Given a set of \( n \) training points \( S = \{z_1, z_2, \ldots, z_n \in Z\} \), we train the predictor by finding parameters \( w \) that minimize the training loss \( \sum_{i=1}^{n} \ell(w, z_i) \), via an iterative optimization procedure (such as stochastic gradient descent) which utilizes one training example \( z_t \in S \) in iteration \( t \), updating the parameter vector from \( wt \) to \( wt+1 \). Then the idealized notion of influence of a particular training example \( z \in S \) on a given test example\(^1\) \( z' \in Z \) is defined as the total reduction in loss on the test example \( z' \) that is induced by the training process whenever the training example \( z \) is utilized, i.e.

\[
\text{TrackInIdeal}(z, z') = \sum_{t: \exists z = z} \ell(w_t, z') - \ell(w_{t+1}, z').
\]

\(^1\)By test example, we simply mean an example whose prediction is being explained. It doesn’t have to be in the test set.

Idealized influence has the appealing property that the sum of the influences of all training examples on a fixed test point \( z' \) is exactly the total reduction in loss on \( z' \) in the training process:

**Lemma 3.1** Suppose the initial parameter vector before starting the training process is \( w_0 \), and the final parameter vector is \( w_T \). Then

\[
\sum_{i=1}^{n} \text{TrackInIdeal}(z_i, z') = \ell(w_0, z') - \ell(w_T, z')
\]

Our treatment above assumes that the iterative optimization technique operates on one training example at a time. Practical gradient descent algorithms almost always operate with a group of training examples, i.e., a minibatch. We cannot extend the definition of idealized influence to this setting, because there is no obvious way to redistribute the loss change across members of the minibatch. In Section 3.3, we will define an approximate version for minibatches.

**Remark 3.2 (Proponents and Opponents)** We will term training examples that have a positive value of influence score as **proponents**, because they serve to reduce loss, and examples that have a negative value of influence score as **opponents**, because they increase loss. In (Koh & Liang, 2017), proponents are called ’helpful’ examples, and opponents called ’harmful’ examples. We chose more neutral terms to make the discussions around mislabelled test examples more natural. (Yeh et al., 2018) uses the terms ’excitory’ and ’inhibitory’, which can be interpreted as proponents and opponents for test examples that are correctly classified, and the reverse if they are misclassified. The distinction arises because the representer approach explains the prediction score and not the loss.

3.2. First-order Approximation to Idealized Influence

Since the step-sizes used in updating the parameters in the training process are typically quite small, we can approximate the change in the loss of a test example in a given iteration \( t \) via a simple first-order approximation:

\[
\ell(w_{t+1}, z') = \ell(w_t, z') + \nabla \ell(w_t, z') \cdot (w_{t+1} - w_t) + O(\|w_{t+1} - w_t\|^2).
\]

Here, the gradient is with respect to the parameters and is evaluated at \( w_t \). Now, if stochastic gradient descent is utilized in training the model, using the training point \( z_t \) at iteration \( t \), then the change in parameters is

\[
w_{t+1} - w_t = -\eta_t \nabla \ell(w_t, z_t),
\]

where \( \eta_t \) is the step size in iteration \( t \). Note that this formula should be changed appropriately if other optimization
methods (such as AdaGrad, Adam, or Newton’s method) are used to update the parameters. The first-order approximation remains valid, however, as long as a small step-size is used in the update.

For the rest of this section we restrict to gradient descent for concreteness. Substituting the formula (2) in (1), and ignoring the higher-order term (which is of the order of $O(\eta_i^2)$), we arrive at the following first-order approximation for the change in the loss:

$$\ell(w_t, z') - \ell(w_{t+1}, z') \approx \eta_t \nabla \ell(w_t, z') \cdot \nabla \ell(w_t, z).$$

For a particular training example $z$, we can approximate the idealized influence by summing up this approximation in all the iterations in which $z$ was used to update the parameters. We call this first-order approximation TrackIn, our primary notion of influence.

$$\text{TrackIn}(z, z') = \sum_{t: z_t = z} \eta_t \nabla \ell(w_t, z') \cdot \nabla \ell(w_t, z).$$

3.3. Extension to Mini-batches

It is common to use mini-batches comprising a number of training data points in each iteration of the optimization process. We can extend the above derivation to mini-batches of size $b \geq 1$. We compute the influence of a mini-batch on the test point $z'$, mimicking the derivation in Section 3.1, and then take its first-order approximation as follows:

$$\text{First-Order Approximation}(B_t, z') = \frac{1}{b} \sum_{z \in B_t} \eta_t \nabla \ell(w_t, z') \cdot \nabla \ell(w_t, z),$$

because the gradient for the mini-batch $B_t$ is $rac{1}{b} \sum_{z \in B_t} \nabla \ell(w_t, z)$. Then, for each training point $z \in B_t$, we attribute the $\frac{1}{b} \eta_t \nabla \ell(w_t, z') \cdot \nabla \ell(w_t, z)$ portion of the influence of $B_t$ on the test point $z'$. Summing up over all iterations $t$ in which a particular training point $z$ was chosen in $B_t$, we arrive at the following definition of TrackIn when mini-batches are used in training:

$$\text{TrackIn}(z, z') = \frac{1}{b} \sum_{t: z_t = z} \eta_t \nabla \ell(w_t, z') \cdot \nabla \ell(w_t, z).$$

Remark 3.3 The derivation suggests a way to measure the goodness of the approximation for a given step: We can check that the change in loss for a step $\ell(w_t, z') - \ell(w_{t+1}, z')$ is approximately equal to First-Order Approximation$(B_t, z')$.

3.4. Random Projection Approximation

Modern deep learning models frequently have a huge number of parameters, making the inner product computations in the first-order approximation of the influence expensive, especially in the case where the influence on a number of different test points needs to be computed. In this situation, we can speed up the computations significantly by using the technique of random projections. This method allows us to pre-compute low-memory sketches of the loss gradients of the training points which can then be used to compute randomized unbiased estimators of the influence on a given test point. The same sketches can be re-used for multiple test points, leading to computational savings. This is a well-known technique (see for example (Woodruff et al., 2014)) and here we give a brief description of how this is done.

Choose a random matrix $G \in \mathbb{R}^{d \times p}$, where $d \ll p$ is a user-defined dimension for the random projections (larger $d$ leads to lower variance in the estimators), whose entries are sampled i.i.d. from $\mathcal{N}(0, \frac{1}{d})$, so that $\mathbb{E}[G^T G] = I$. We compute the following sketch: in iteration $t$, compute and save $\eta_t G \nabla \ell(w_t, z_i)$. Then given a test point $z'$, the dot product $(\eta_t G \nabla \ell(w_t, z_i)) \cdot (G^T \nabla \ell(w_t, z'))$ is an unbiased estimator of $\eta_t \nabla \ell(w_t, z_i) \cdot (\nabla \ell(w_t, z'))$, and can thus be substituted in all influence computations.

3.5. Practical Heuristic Influence via Checkpoints

The method described so far does not scale to typically used long training processes since it involves keeping track of the parameters, as well as training points used, at each iteration: effectively, in order to compute the influence, we need to replay the training process, which is obviously impractical. In order to make the method practical, we employ the following heuristic. It is common to store checkpoints (i.e. the current parameters) during the training process at regular intervals. Suppose we have $k$ checkpoints $w_{t_1}, w_{t_2}, \ldots, w_{t_k}$ corresponding to iterations $t_1, t_2, \ldots, t_k$. We assume that between checkpoints each training example is visited exactly once. (This assumption is only needed for an approximate version of Lemma 3.1; even without this, TrackInCP is a useful measure of influence.) Furthermore, we assume that the step size is kept constant between checkpoints, and we use the notation $\eta_i$ to denote the step size used between checkpoints $i - 1$ and $i$. While the first-order approximation of the influence needs the parameter vector at the specific iteration where a given training example is visited, since we don’t have access to the parameter vector, we simply approximate it with the first checkpoint parameter vector after it. Thus, this heuristic results in the following formula:

$$\text{TrackInCP}(z, z') = \sum_{i=1}^{k} \eta_i \nabla \ell(w_{t_i}, z) \cdot \nabla \ell(w_{t_i}, z').$$
Remark 3.4 (Low Latency Implementation) We can use an approximate nearest neighbors technique to quickly identify influential examples for a specific prediction. The idea is to pre-compute the training loss gradients at the various checkpoints (possibly using the random projection trick to reduce space). Then, we concatenate the loss gradients for a given training point \( z \) (i.e., \( \ell(w_{t_1}, z), \ell(w_{t_2}, z) \ldots \ell(w_{t_n}, z) \)) together into one vector. This can be then loaded into an approximate nearest neighbor library (e.g. (Bernhardsson, 2018)). During analysis, we can do the same for a test example—the gradient calls for the different checkpoints can be done in parallel. We then invoke nearest neighbor search. The nearest neighbor library then performs the computation implicit in Equation 4.

Remark 3.5 (Handling Variations of Training) In our derivation of TrackIn we have assumed a certain form of training. In practice, there are likely to be differences in optimizers, learning rate schedules, the handling of mini-batches etc. It should be possible to redo the derivation of TrackIn to handle these differences. Also, we expect the practical form of TrackInCP to remain the same across these variations.

Remark 3.6 (Selecting Checkpoints) In the application of TrackInCP, we choose checkpoints at epoch boundaries, i.e., between checkpoints, each training example is visited exactly once. However, it is possible to be smarter about how checkpoints are chosen: Generally, it makes sense to sample checkpoints at points in the training process where there is a steady decrease in loss, and to sample more frequently when the rate of decrease is higher. It is worth avoiding checkpoints at the beginning of training when loss fluctuates. Also, checkpoints that are selected after training has converged add little to the result, because the loss gradients here are tiny. Relatedly, computing TrackInCP with just the final model could result in noisy results.

4. Evaluations

In this section we compare TrackIn with influence functions (Koh & Liang, 2017) and the representer point selection method (Yeh et al., 2018). Brief descriptions of these two methods can be found in the supplementary material. We also compare practical implementations of TrackIn against an idealized version.

4.1. Conceptual Comparison of the Methods

Beyond the simplicity of TrackIn compared to the influence functions or representer methods, a primary point of distinction TrackIn is that unlike the other two methods, it makes no optimality assumptions on the trained classifier. The other two methods are only defined if the parameters of the trained classifier satisfy, at the very least, some local optimality conditions. This point is crucial to meaningfully deploy any method in practice, since modern deep learning models are rarely, if ever, trained to even moderate-precision convergence. Regardless, we do compare our approach against both of these approaches in the subsequent sections.

4.2. Evaluation Approach

Real world datasets can contain mislabelled examples, that can harm model performance. One application of any training data influence computation technique is to identify mislabelled examples automatically. This has been used as a form of evaluation (see Section 4.1 (Yeh et al., 2018) and Section 5.4 of (Koh & Liang, 2017)). The idea is to measure self-influence, i.e., the influence of a training point on its own loss, i.e., the training point \( z \) and the test point \( z' \) in Equation 3 are identical.

Incorrectly labelled examples are likely to be strong proponents (recall terminology in Section 3.1) for themselves. Strong, because they are outliers, and proponents because they would tend to reduce loss (with respect to the incorrect label). Therefore, when we sort training examples by decreasing self-influence, an effective influence computation method would tend to rank mislabelled examples in the beginning of the ranking. We use the fraction of mislabelled data recovered for different prefixes of the rank order as our evaluation metric; higher is better.

To simulate the real world mislabelling errors, we first trained a model on correct data. Then, for 10% of the training data, we changed the label to the highest scoring incorrect label. We then attempt to identify mislabelled examples as discussed above.

4.3. CIFAR-10

In this section, we work with ResNet-56 (He et al., 2016) trained on the CIFAR-10 (Krizhevsky et al., 2009). The model on the original dataset has 93.4% test accuracy.  

4.3.1. Identifying Mislabelled Examples

Recall the evaluation set up and metric in Section 4.2. Training on the mislabelled data reduces test accuracy from 93.4% to 87.0% (train accuracy is 99.6%). We compare TrackIn with influence functions (Section A.1) and the representer point selection method (Section A.2).

For influence functions, it is prohibitively expensive to compute the Hessian for the whole model, so work with parameters in the last layer, essentially considering the layers below...
the last as frozen. This mimics the set up in Section 5.1 of (Koh & Liang, 2017). Given that CIFAR-10 only has 50K training examples, we directly compute inverse hessian by definition.

For representer points, we fine-tuned the last layer with line-search, which requires the full batch to find the stationary point and use \( |\alpha_{ij}| \) as described in Section 4.1 of (Yeh et al., 2018) to compare with self-influence. We use TrackInCP with only the last layer. We sample every 30 checkpoints starting from the 30th checkpoint; every checkpoint was at a epoch boundary. The right hand side of Figure 1a shows that TrackInCP identifies a larger fraction of the mislabeled training data (y-axis) regardless of the fraction of the training data set that is examined (x-axis). For instance, TrackIn recovers more than 80% of the mislabelled data in the first 20% of the ranking, whereas the other methods recover less than 50% at the same point. Furthermore, we show that fixing the mislabelled data found within a certain fraction of the training data, results in a larger improvement in test accuracy for TrackIn compared to the other methods (see the plot on the left hand side of Figure 1a). We also show that weighting checkpoints equally yield similar results. This provides support to ignore learning rate for implementation simplification.

4.3.2. EFFECT OF DIFFERENT CHECKPOINTS ON TrackIn SCORES

Next, we discuss the contributions of the different checkpoints to the scores produced by TrackIn; recall that TrackIn computes a weighted average across checkpoints (see Equation 4). We find that different checkpoints contain different information. We identify the number of mislabelled examples from each class (the true class, not the mislabelled class) within the first 10% of the training data in Fig. 9 (in the supplementary material), we show results for the 30th, 150th and 270th checkpoint. We find that the mix of classes is different between the checkpoints. The 30th checkpoint has a larger fraction (and absolute number) of mislabelled deer and frogs, while the 150th emphasizes trucks. This is likely because the model learns to identify different classes at different points in training process, highlighting the importance of sampling checkpoints.

4.4. MNIST

In this section, we work on the MNIST digit classification task. We use a model with 3 hidden layers and 240K parameters. This model has 97.55% test accuracy. Because the model is smaller than the Resnet model we used for CIFAR-10, we can perform a slightly different set of comparisons. First, we are able to compute approximate influence for each training step (Section 3.3), and not just heuristic influence using checkpoints. Second, we can apply TrackIn and the influence functions method to all the model parameters, not just the last layer.

Since we have a large number of parameters, we resort to a randomized sketching based estimator of the influence whose description can be found in the supplementary material. In our experiments, this model would sometimes not converge, and there was significant noise in the influence scores, which are estimating a tiny effect of excluding one training point at a time. To mitigate these issues, we pick lower learning rates, and use larger batches to reduce variance, making the method time-intensive.

4.4.1. VISUAL INSPECTION OF PROPONENTS AND OPPONENTS

We eyeball proponents and opponents of a random sample of test images from MNIST test set. We observe that TrackInCP and representer consistently find proponents visually similar to test examples. Although, the opponents picked by representer are not always visually similar to test example (the opponent ‘7’ in Figure 2a and ‘5’ and ‘8’s in Figure 2b). In contrast, TrackInCP seems to pick pixel-wise similar opponents.
4.4.2. IDENTIFYING MISLABELLED EXAMPLES

Recall the evaluation set up and metric in Section 4.2. We train on MNIST mislabelled data as described there. After 140 epochs, it achieves accuracy of 89.94% on mislabelled train set, and 89.95% on test set. As in Section 4.3.1 we compare against the influence functions method and the representer point method. Similar to CIFAR-10, TrackIn outperforms the other two methods and retrieves a larger fraction of mislabelled examples for different fractions of training data inspected (Figure 1b). Furthermore, as expected, batch approximate TrackIn is able to recover mislabelled examples faster than heuristic TrackInCP (we use every 30th checkpoint, starting from 20th checkpoint), but not by a large margin.

Next, we evaluate the effects of our various approximations. We use the same 3-layer model architecture, but with the correct MNIST dataset. The model has 97.55% test set accuracy on test set, and 99.30% train accuracy.

4.4.3. EFFECT OF THE FIRST-ORDER APPROXIMATION

We now evaluate the effect of the first-order approximation (described in Sections 3.3 and 3.2). By Remark 3.3, we would like the total first-order influence at a step First-Order Approximate Influence($B_t, z'$) to approximate the change in loss at the step $\ell(w_t, z') - \ell(w_t, z')$. Figure 8 (in the supplementary material) shows the relationship between the two quantities; every point corresponds to one parameter update step for one test point. We consider 100 random test points. The overall Pearson correlation between the two quantities is 0.978, which is sufficiently high.

4.4.4. EFFECT OF CHECKPOINTS

We now discuss the approximation from Section 3.5, i.e., the effect of using checkpoints. We compute the correlation of the influence scores of 100 test points using TrackInCP with different checkpoints against the scores from the first-order approximation TrackIn. As discussed in Remark 3.6, we find that selecting checkpoints with high loss reduction, are more informational than selecting same number of evenly spaced checkpoints. This is because in later checkpoints the loss flattens, hence, the loss gradients are small. Figure 3 shows TrackInCP with just one checkpoint from middle correlates more than the last checkpoint with TrackIn scores. Consequently, TrackInCP with more checkpoints improves the correlation, more so if the checkpoints picked had high loss reduction rates.

4.4.5. EFFECT OF RANDOM PROJECTIONS

As mentioned in Section 3.4, gradient matrices can be projected onto much smaller dimensions to speed-up computation and to save space. We study the effect of this approximation, by layering this approximation on top of the checkpoint approximation TrackInCP (we use six checkpoints). Figure 3 also shows correlations for random projections of different numbers of dimensions. No projection has a 0.889 median correlation, while using projection of dimension 1000 shifts the median to 0.880 while providing 240:1 compression of the gradient matrices.

5. Applications

We apply TrackIn to a regression problem (Section 5.1) a text problem (Section 5.2) and an computer vision problem (Section 5.3) to demonstrate its versatility. The last of these use cases is on a ResNet-50 model trained on the (large)
We apply TrackIn and find that the high self-influence examples come from densely populated locations, where memorization would hurt model performance (Figure 4b).

Figure 4. TrackIn on California housing prices dataset.

Imagenet dataset, demonstrating that TrackIn scales.

5.1. California Housing Prices

We study TrackIn on a regression problem using California housing prices dataset (Pace & Barry, 1997). We used a 80:20 train-test split and trained a regression model with 3 hidden layers with 168K parameters, using Adam optimizer minimizing MSE for 200 epochs. The model achieves explained variance of 0.70 on test set, and 0.72 on train set. We use every 20th checkpoint to get TrackIn influences.

The notion of comparables in real estate refers to recently sold houses that are similar to a home in location, size, condition and features, and are therefore indicative of the home’s market value. We can use TrackInCP to identify model-based comparables, by examining the proponents for certain predictions. For instance, we could study proponents for houses in the city of Palo Alto, a city in the Bay Area known for expensive housing. We find that the proponents are drawn from other areas in the Bay Area, and the cities of Sacramento, San Francisco and Los Angeles (Figure 4a). One of the influential examples lies on the island of Santa Catalina, also known for expensive housing.

We also study self-influences of training examples (see Section 4.2 for the definition of self-influence). High self-influence is more likely to be indicative of memorization. We find that the high self influence examples come from densely populated locations, where memorization is reasonable, and conversely, low self-influence ones comes from sparsely populated areas, where memorization would hurt model performance (Figure 4b).

5.2. Text Classification

We apply TrackIn on the DBPedia ontology dataset introduced in (Zhang et al., 2015). The task is to predict the ontology with title and abstract from Wikipedia. The dataset consists of 560K training examples and 70K test examples equally distributed among 14 classes. We train a Simple Word-Embedding Model (SWEM) (Shen et al., 2018) for 60 epochs and use the default parameters of sentencepiece library as tokenizer (Kudo & Richardson, 2018) and achieve 95.5% on both training and test. We apply TrackInCP and sample 6 evenly spaced checkpoints and the gradients are taken with respect to the last fully connected layer.

Table 1 shows the top 3 opponents for one test example (Manuel Azana); we filter misclassified training examples from the list to find a clearer pattern. Misclassified examples have high loss, and therefore high training loss gradient, and are strong proponents/opponents for different test examples, and are thus not very discriminative.) The list of opponents provide insight about data introducing correlation between politicians and artists.

5.3. Imagenet Classification

Real world applications tend to be large in data size. Methods approximating influence of training data should be able to scale at least linearly as the data size increases. It seems difficult to scale influence functions or representer methods to ImageNet. However, using a special random projection idea it becomes possible to apply TrackIn on the fully connected layer of ResNet-50 trained on Imagenet (Deng et al., 2009)3, which consists of 1.28M training examples with 1000 classes. The 30th, 60th, and 90th checkpoints are used for TrackInCP and we project the gradients to a vector of size 1472. The random projection idea relies on the fact that for fully-connected layers, the gradient of the loss w.r.t. the weights for the layer is a rank 1 matrix. Thus, TrackIn involves computing the dot (Hadamard) product of two rank 1 matrices, for which much faster random projection based estimators than the ones described in Section 3.4 exist. Details are in the supplementary material.

We show three proponents and three opponents for five examples in figure 5. We filtered out misclassified examples as we did for text classification. A few quick observations:

(i) The proponents are mostly images from the same label.
(ii) In the first row of figure 5, the style of the microphone in the test example is different from the top proponents, perhaps augmenting the data with more images that resemble the test one can fix the misclassification.
(iii) For the correctly classified test examples, the opponents give us an idea which examples would confuse the model (for the church, there are castles, for the bostonbull there are french bulldogs, for the wheel there are loupes and spotlights, and for the chameleon there is a closely related animal (agama) but there are also broccoli and jackfruits.

---

3 The model is trained for 90 epochs and achieves 73% top-1 accuracy.
Table 1. Opponents for text classification on DBPedia. All examples shown have the same label and prediction. Proponents can be found in Appendix.

| Example | OfficeHolder |
|---------|--------------|
| **Manuel Azaña** Manuel Azaña Díaz (Alcalá de Henares January 10 1880–Montauban November 3 1940) was the first Prime Minister of the Second Spanish Republic (1931–1933) and later served again as Prime Minister (1936) and then as the second and last President of the Republic (1936–1939). The Spanish Civil War broke out while he was President. With the defeat of the Republic in 1939 he fled to France resigned his office and died in exile shortly afterwards. |

| Opponents | Artist |
|-----------|--------|
| **Mikołaj Rej** Mikołaj Rej or Mikołaj Rey of Nagłowice (February 4 1505 – between September 8 and October 5 1569) was a Polish poet and prose writer of the emerging Renaissance in Poland as it succeeded the Middle Ages as well as a politician and musician. He was the first Polish author to write exclusively in the Polish language and is considered (with Biernat of Lublin and Jan Kochanowski) to be one of the founders of Polish literary language and literature. |

| Opponents | Artist |
|-----------|--------|
| **Justin Jeffre** Justin Paul Jeffre (born on February 25 1973) is an American pop singer and politician. A long-time resident and vocal supporter of Cincinnati Jeffre is probably best known as a member of the multi-platinum selling boy band 98 Degrees. Before shooting to superstardom Jeffre was a student at the School for Creative and Performing Arts in Cincinnati. It was there that he first became friends with Nick Lachey. The two would later team up with Drew Lachey and Jeff Timmons to form 98 Degrees. |

| Opponents | Artist |
|-----------|--------|
| **David Kitt** David Kitt (born 1975 in Dublin Ireland) is an Irish musician. He is the son of Irish politician Tom Kitt. He has released six studio albums to date: Small Moments The Big Romance Square 1 The Black and Red Notebook Not Fade Away and The Nightsaver. |

Figure 5. TrackIn applied on Imagenet. Each row starts with the test example followed by three proponents and three opponents. The test image in the first row is classified as band-aid and is the only misclassified example. (af-chameleon: african-chameleon, fr-bulldog: french-bulldog)
References

Bernhardsson, E. Annoy: Approximate Nearest Neighbors in C++/Python, 2018. URL https://pypi.org/project/annoy/. Python package version 1.13.0.

Bien, J. and Tibshirani, R. Prototype selection for interpretable classification. The Annals of Applied Statistics, pp. 2403–2424, 2011.

Cook, R. D. and Weisberg, S. Residuals and influence in regression. New York: Chapman and Hall, 1982.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pp. 248–255. Ieee, 2009.

Ghorbani, A. and Zou, J. Data shapley: Equitable valuation of data for machine learning. In International Conference on Machine Learning, pp. 2242–2251, 2019.

Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. Jama, 316(22): 2402–2410, 2016.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770–778, 2016.

Jia, R., Dao, D., Wang, B., Hubis, F. A., Hynes, N., Gürel, N. M., Li, B., Zhang, C., Song, D., and Spanos, C. J. Towards efficient data valuation based on the shapley value. In The 22nd International Conference on Artificial Intelligence and Statistics, pp. 1167–1176, 2019.

Kim, B., Rudin, C., and Shah, J. A. The bayesian case model: A generative approach for case-based reasoning and prototype classification. In Advances in Neural Information Processing Systems, pp. 1952–1960, 2014.

Kim, B., Khanna, R., and Koyejo, O. O. Examples are not enough, learn to criticize! criticism for interpretability. In Advances in neural information processing systems, pp. 2280–2288, 2016.

Koh, P. W. and Liang, P. Understanding black-box predictions via influence functions. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 1885–1894, 2017.

Krizhevsky, A. et al. Learning multiple layers of features from tiny images. 2009.

Kudo, T. and Richardson, J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 66–71, 2018.

Lundberg, S. M. and Lee, S.-I. A unified approach to interpreting model predictions. In Advances in neural information processing systems, pp. 4765–4774, 2017.

Owen, A. B. Sobol’indices and shapley value. SIAM/ASA Journal on Uncertainty Quantification, 2(1):245–251, 2014.

Owen, A. B. and Prieur, C. On shapley value for measuring importance of dependent inputs. SIAM/ASA Journal on Uncertainty Quantification, 5(1):986–1002, 2017.

Pace, R. K. and Barry, R. Sparse spatial autoregressions. Statistics & Probability Letters, 33(3):291–297, 1997.

Pearlmutter, B. A. Fast exact multiplication by the hessian. Neural Computation, 6:147–160, 1994.

Ribeiro, M. T., Singh, S., and Guestrin, C. “ Why should i trust you?” Explaining the predictions of any classifier. In Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pp. 1135–1144, 2016.

Schölkopf, B., Herbrich, R., and Smola, A. J. A generalized representer theorem. In International conference on computational learning theory, pp. 416–426. Springer, 2001.

Shen, D., Wang, G., Wang, W., Min, M. R., Su, Q., Zhang, Y., Li, C., Henao, R., and Carin, L. Baseline needs more love: On simple word-embedding-based models and associated pooling mechanisms. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 440–450, 2018.

Sundararajan, M., Taly, A., and Yan, Q. Axiomatic attribution for deep networks. In Proceedings of the 34th International Conference on Machine Learning-Volume 70, pp. 3319–3328, 2017.

Woodruff, D. P. et al. Sketching as a tool for numerical linear algebra. Foundations and Trends® in Theoretical Computer Science, 10(1–2):1–157, 2014.

Yeh, C.-K., Kim, J., Yen, I. E.-H., and Ravikumar, P. K. Representer point selection for explaining deep neural networks. In Advances in Neural Information Processing Systems, pp. 9291–9301, 2018.

Zhang, X., Zhao, J., and LeCun, Y. Character-level convolutional networks for text classification. In Advances in neural information processing systems, pp. 649–657, 2015.
A. Description of Influence Functions and Representer Point Methods

A.1. Influence Functions

Koh & Liang (2017) proposed using the idea of Influence functions (Cook & Weisberg, 1982) to measure the influence of a training point on a test example. Specifically, they use optimality conditions for the model parameters to mimic the effect of perturbing single training example:

$$\text{Inf}(z, z') = -\nabla_{\hat{w}}\ell(\hat{w}, z') \cdot H^{-1}_{\hat{w}} \cdot \nabla_{\hat{w}}\ell(\hat{w}, z).$$ \hfill (5)

Here, $H_{\hat{w}} = \frac{1}{n} \sum_{i=1}^n \nabla^2 \ell(\hat{w}, z_i)$ is the Hessian. As pointed out by Koh & Liang (2017), for large deep learning models with massive training sets, the inverse Hessian computation is costly and complex. This technique also assumes that the model is at convergence so that the optimality conditions hold.

A.1.1. Scalable implementation via randomized sketching

It becomes infeasible to compute the inverse Hessian when the number of parameters is very large, as is common in modern deep learning models. To mitigate this issue we compute randomized estimators of $H_{\hat{w}}^{-1} \nabla_{\hat{w}}\ell(\hat{w}, z')$ via a sketch of the inverse Hessian in the form of the product $H_{\hat{w}}^{-1}G^\top$ where $G$ is the same kind of random matrix as in Section 3.4. The product $[H_{\hat{w}}^{-1}G^\top][G \nabla_{\hat{w}}\ell(\hat{w}, z')]$ is then an unbiased estimator of $H_{\hat{w}}^{-1}\nabla_{\hat{w}}\ell(\hat{w}, z')$. Note that the sketch takes only $O(dp)$ memory rather than $O(p^2)$ that the inverse Hessian would take. We compute the sketch by solving the optimization problem $\min_S \|H_{\hat{w}}S - G^\top\|^2_F$, via a customized stochastic gradient descent procedure based on the formula

$$\nabla_S \|H_{\hat{w}}S - G^\top\|^2_F = 2H_{\hat{w}}(H_{\hat{w}}S - G^\top).$$

This customized stochastic gradient descent procedure uses the following stochastic gradient computed using two independently chosen minibatches of examples $B_1, B_2$ instead of the customary one:

$$2\left[\frac{1}{|B_1|} \sum_{z \in B_1} \nabla^2 \ell(\hat{w}, z)\right] - \frac{1}{|B_2|} \sum_{z \in B_2} \nabla^2 \ell(\hat{w}, z) - G^\top].$$ \hfill (6)

Note that $\mathbb{E}\left[\frac{1}{|B_1|} \sum_{z \in B_1} \nabla^2 \ell(\hat{w}, z)\right] = H_\phi$ and $\mathbb{E}\left[\frac{1}{|B_2|} \sum_{z \in B_2} \nabla^2 \ell(\hat{w}, z)\right] = H_{\phi}$, and since $B_1$ and $B_2$ are independently chosen, we conclude that the expectation of the quantity in (6) is indeed $2H_\phi(H_\phi S - G^\top)$ as required. Note that (6) can be computed using Hessian-vector products, which can be computed easily using the Pearlmutter trick (Pearlmutter, 1994).

A.2. Representer Point Selection

The second method is proposed in (Yeh et al., 2018) and is based on the representer point theorem (Schölkopf et al., 2001). The method decomposes the logits for any test point into a weighted combination of dot products between the representation of the test point at the top layer of a neural network and those of the training points; this is effectively a kernel method. The weights in the decomposition capture the influences of that training points.

Specifically, consider a neural network model with fitted parameters into $\{w_1, w_2\}$, where $w_2$ is the matrix of parameters that produces the logits from the input representation (i.e. the top layer weights) and $w_1$ are the remaining parameters. To meet the conditions of the representer theorem, the final layer of the model is tuned by adding a term L2 regularization term $\lambda \|w_2\|^2$ to the loss and training the model to convergence. This optimization produces a new set of parameters $w_2'$ for the last layer, resulting in a new model with parameters $w' = \{w_1, w_2'\}$. Then the influence of a training example $z$ on a test example $z'$ is a $k$-dimensional vector (one element per class) given by

$$\text{Rep}(z, z') = -\frac{1}{2\lambda n} (f(w_1, z) \cdot f(w_1, z'))\partial_{\phi(w', z')} \ell(w', z').$$ \hfill (7)

Here, $f(w_1, z)$ is the input representation, i.e. the outputs of the last hidden layer, and $\phi(w', z) = w'_f f(w_1, z)$ are the logits. The L2 regularization requires a complex, memory-intensive line search, and results in a model different from the original one, possibly resulting in influences that are unfaithful to the original model. Conceptually, it is also not clear how to study the influence that flows via the parameters in lower layers—computing a stationary point is harder in this situation. Furthermore, both the influence functions approach and TrackIn could be used to explain the influence of a training example on the loss of a test example or its prediction score. In contrast, it is unclear how to use the representer point method to explain loss on a test example.

B. A Visual Inspection of Proponents and Opponents for CIFAR

We now consider the same training procedure in Section 4.1 but on the regular CIFAR-10 dataset. We show the top 5 proponent and opponent examples of an image from the test set and compare the three methods qualitatively in Figures 6 and 7. All three methods retrieved mostly cats as positive examples and dogs as negative examples, but TrackIn seems more consistent on the types of cats and dogs. For the mis-classified automobile, proponents of TrackIn pick up automobiles of a similar variety type.
Figure 6. CIFAR-10 results: Proponents and opponents examples of a correctly classified cat for influence functions, representer point, and TrackIn. (Predicted class in brackets)

Figure 7. CIFAR-10 results: Proponents and opponents examples of an incorrectly classified automobile for influence functions, representer point, and TrackIn. (Predicted class in brackets)

C. Fast Random Projections for Gradients of Fully-Connected Layers

Suppose we have a fully connected layer in the neural network with a weight matrix \( W \in \mathbb{R}^{m \times n} \), where \( m \) is the number of units in the input to that layer, and \( n \) is the number of units in the output of the layer. For the purpose of TrackIn computations, it is possible to obtain a random projection of the gradient w.r.t. \( W \) into \( d \) dimensions with time and space complexity \( O((m + n) \cdot \sqrt{d}) \) rather than the naive \( O(mnd) \) complexity that the standard random projection needs.

To formalize this, let us represent the layer as performing the following computation: \( y := Wx \) where \( x \in \mathbb{R}^{n} \) is the input to the layer, and \( y \) is the vector of pre-activations (i.e. the value fed into the activation function). Now suppose we want to compute the gradient of some function \( f \) (e.g. loss, or prediction score) of the output of the layer, i.e. we want to compute \( \nabla_{W}(f(Wx)) \). A simple application of the chain rule shows gives the following formula for the gradient:

\[
\nabla_{W}(f(Wx)) = \nabla_{y} f(y) x^{T}.
\]

In particular, note that the gradient w.r.t. \( W \) is rank 1. This property is very useful for TrackIn since it involves computations of the form \( \nabla_{W}(f(Wx)) \cdot \nabla_{W}(f'(Wx')) \), where \( f' \) is another function and \( x' \) is another input. Note that for \( y' = Wx' \), we have

\[
\nabla_{W}(f(Wx)) \cdot \nabla_{W}(f'(Wx')) = (\nabla_{y} f(y) x^{T}) \cdot (\nabla_{y'} f'(y') x'^{T}) = (\nabla_{y} f(y) \cdot \nabla_{y'} f'(y'))(x \cdot x').
\]

The final expression can be computed in \( O(m + n) \) time by computing the two dot products \( (\nabla_{y} f(y) \cdot \nabla_{y'} f'(y')) \) and \( (x \cdot x') \) separately and then multiplying them. This is much
faster than the naive dot product of the gradients, which takes $O(mn)$ time.

This can already speed up TrackIn computations. We can also save on space by randomly projecting $\nabla_y f(y)$ and $x$ separately, but unfortunately this doesn’t seem to be amenable to fast nearest-neighbor search. If we want to use fast nearest-neighbor search, we will need to use random projections in the following manner which also exploits the rank-1 property. To project into $d$ dimensions, we can use two independently chosen random projection matrices $G_1 \in \mathbb{R}^{\sqrt{d} \times m}$ and $G_2 \in \mathbb{R}^{\sqrt{d} \times n}$, with $E[G_1 G_1^\top] = E[G_2 G_2^\top] = I$, and compute

$$G_1 \nabla_y f(y) x^\top G_2^\top \in \mathbb{R}^{\sqrt{d} \times \sqrt{d}},$$

which can be flattened to a $d$-dimensional vector. Note that this computation requires time and space complexity $O((m + n) \cdot \sqrt{d})$. Furthermore, since $G_1$ and $G_2$ are chosen independently, it is easy to check that

$$E[(G_1 \nabla_y f(y) x^\top G_2^\top) \cdot (G_1 \nabla_y' f'(y') x'^\top G_2^\top)] = (\nabla_y f(y) x^\top) \cdot (\nabla_y' f'(y') x'^\top),$$

so the randomized dot-product is unbiased.

D. Additional Results

This section contains charts and images that support discussions in the main body of the paper.
TrackIn

Figure 8. Comparison of change in loss at all training steps and TrackIn influences at those steps for 100 test examples from MNIST dataset. This measures the quality of the first-order approximation—see Section 4.4.3.

Table 2. Proponents for text classification on DBPedia—see Section 5.2. All examples shown have the same label and prediction.

| Example | OfficeHolder |
|---------|--------------|
| Manuel Azaña Manuel Azaña Díaz (Alcalá de Henares January 10 1880 – Montauban November 3 1940) was the first Prime Minister of the Second Spanish Republic (1931–1933) and later served again as Prime Minister (1936) and then as the second and last President of the Republic (1936–1939). The Spanish Civil War broke out while he was President. With the defeat of the Republic in 1939 he fled to France resigned his office and died in exile shortly afterwards. | |

| Proponents | OfficeHolder |
|------------|--------------|
| Annemarie Huber-Hotz Annemarie Huber-Hotz (born 16 August 1948 in Baar Zug) was Federal Chancellor of Switzerland between 2000 and 2007. She was nominated by the FDP for the office and elected on 15 December 1999 after four rounds of voting. The activity is comparable to an office for Minister. The Federal Chancellery with about 180 workers performs administrative functions relating to the co-ordination of the Swiss Federal government and the work of the Swiss Federal Council. | |

| Proponents | OfficeHolder |
|------------|--------------|
| Jose Manuel Restrepo Velez Jose Manuel Restrepo Velez (30 December 1781 – 1 April 1863) was an investigator of Colombian flora political figure and historian. The Orchid genus Restrepia was named in his honor. Restrepo was born in the town of Envigado Antioquia in the Colombian Mid-west. He graduated as a lawyer from the Colegio de San Bartolomé in the city of Santa Fe de Bogotá. He later worked as Secretary for Juan del Corral and Governor Dionisio Tejada during their dictatorial government over Antioquia. | |

| Proponents | OfficeHolder |
|------------|--------------|
| K. C. Chan Professor Ceajer Ka-keung Chan (Traditional Chinese: 陳家強) SBS JP (born 1957) also referred as KC Chan is the Secretary for Financial Services and the Treasury in the Government of Hong Kong. He is also the ex officio chairman of the Kowloon-Canton Railway Corporation and an ex officio member of the Hong Kong International Theme Parks Board of Directors. | |
Figure 9. Number of identified mislabeled examples by class for three checkpoints within the top 10% of ranking by self-influence. Different checkpoints highlight different labels—see Section 4.3.2.
Figure 10. MNIST (Section 4.4): Proponents and opponents examples of a correctly classified images for TrackIn, representer point, and influence functions. (Predicted class in brackets).