Estimating Training Data Influence by Tracing Gradient Descent

Garima[*]
Google
pruthi@google.com

Frederick Liu[*]
Google
frederickliu@google.com

Satyen Kale
Google
satyenkale@google.com

Mukund Sundararajan [†]
Google
mukunds@google.com

Abstract

We introduce a method called `TracIn` that computes the influence of a training example on a prediction made by the model. The idea is to trace 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 `TracIn` via: (a) a first-order gradient approximation to the exact computation, (b) saved checkpoints of standard training procedures, and (c) cherry-picking layers of a deep neural network. In contrast with previously proposed methods, `TracIn` is *simple* to implement; all it needs is the ability to work with gradients, checkpoints, and loss functions. The method is *general*. It applies to any machine learning model trained using stochastic gradient descent or a variant of it, agnostic of architecture, domain and task. We expect the method to be widely useful within processes that study and improve training data. Code is available at [1].

1 Motivation

Deep learning has been used to solve a variety of real-world problems. A common form of machine learning is supervised learning, where the model is trained on *labelled* data. 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 4.1. 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*. We propose a method called `TracIn` for computing this influence, provide a scalable implementation, and evaluate the method experimentally.

2 Related Work

[2, 3] tackle influential training examples in the context of deep learning. We discuss these methods in detail in Section 4.4.

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, [4, 5, 6] identify the influence of features on an individual prediction. [7, 8] identify the influence of features on the

[*]Equal contribution.
[†]Corresponding author.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

overall accuracy (loss) of the model. [9, 10] identify the influence of training examples on the overall accuracy of the model. [11], a technique closely related to TracIn, identifies the influence of training examples on the overall loss by tracing the training process while TracIn identifies the influence on the loss of a test point. The key trick in [11] is to use a certain hessian of the model parameters to trace the influence of a training point through minibatches in which it is *absent*. This trick is also potentially useful in implementing idealized version of TracIn. However, idealized TracIn requires the test/inference point to be known at training time and is therefore impractical, making the trick less relevant to the problem we study. TracInCP, a practical implementation, leverages checkpoints to replay the training process. Checkpoint ensembling is a widely used technique in machine translation [12], semi-supervised learning [13] and knowledge distillation [14] which provide intuition on why TracIn performs better than other methods.

3 The Method

In this section we define TracIn. TracIn is inspired by the *fundamental theorem of calculus*. The fundamental theorem of calculus decomposes the difference between a function at two points using the gradients along the path between the two points. Analogously, TracIn decomposes the difference between the loss of the test point at the end of training versus at the beginning of training along the path taken by the training process.[3]

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 $\ell : \mathbb{R}^p \times Z \to \mathbb{R}$; 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 w_t to w_{t+1}. Then the idealized notion of influence of a particular **training example** $z \in S$ on a given **test example**[4] $z' \in Z$ 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. $\texttt{TracInIdeal}(z, z') = \sum_{t:\, z_t = z} \ell(w_t, z') - \ell(w_{t+1}, z')$

Recall that TracIn was inspired by the fundamental theorem of calculus, which has the property that the integration of the gradients of a function between two points is equal to the difference between function values between the two points. Analogously, 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 $\sum_{i=1}^{n} \texttt{TracInIdeal}(z_i, z') = \ell(w_0, z') - \ell(w_T, z')$*

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 3.2, 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*

[3] With the minor difference that the training process is a discrete process, whereas the path used within the fundamental theorem is continuous.

[4] By test example, we simply mean an example whose prediction is being explained. It doesn't have to be in the test set.

*a negative value of influence score as **opponents**, because they increase loss. In [2], 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. [3] 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, and Extension to Minibatches

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 η_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 change in parameters formula in the first-order approximation, and ignoring the higher-order term (which is of the order of $O(\eta_t^2)$), we arrive at the following first-order approximation for the change in the loss $\ell(w_t, z') - \ell(w_{t+1}, z') \approx \eta_t \nabla\ell(w_t, z') \cdot \nabla\ell(w_t, z_t)$. 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 `TracIn`, our primary notion of influence: $\texttt{TracIn}(z, z') = \sum_{t:\ z_t = z} \eta_t \nabla\ell(w_t, z') \cdot \nabla\ell(w_t, z)$.

To handle minibatches of size $b \geq 1$, we compute the influence of a minibatch on the test point z', mimicking the derivation in Section 3.1, and then take its first-order approximation: $\text{First-Order Approximation}(B_t, z') = \frac{1}{b} \sum_{z \in B_t} \eta_t \nabla\ell(w_t, z') \cdot \nabla\ell(w_t, z)$, because the gradient for the minibatch B_t is $\frac{1}{b} \sum_{z \in B_t} \nabla\ell(w_t, z)$. Then, for each training point $z \in B_t$, we attribute the $\frac{1}{b} \cdot \eta_t \nabla\ell(w_t, z') \cdot \nabla\ell(w_t, z)$ 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 `TracIn` with minibatches: $\texttt{TracIn}(z, z') = \frac{1}{b} \sum_{t:\ z \in B_t} \eta_t \nabla\ell(w_t, z') \cdot \nabla\ell(w_t, z)$.

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').*

3.3 Practical Heuristic Influence via Checkpoints

The method described so far does not scale to typically used long training processes since it involves tracing 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, `TracInCP` is a useful measure of influence.) Furthermore, we assume that the step size is kept constant between checkpoints, and we use the notation η_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[5]:

$$\texttt{TracInCP}(z, z') = \sum_{i=1}^{k} \eta_i \nabla \ell(w_{t_i}, z) \cdot \nabla \ell(w_{t_i}, z') \tag{1}$$

Remark 3.4 (Handling Variations of Training) *In our derivation of* `TracIn` *we have assumed a certain form of training. In practice, there are likely to be differences in optimizers, learning rate schedules, the handling of minibatches etc. It should be possible to redo the derivation of* `TracIn` *to handle these differences. Also, we expect the practical form of* `TracInCP` *to remain the same across these variations.*

Remark 3.5 (Counterfactual Interpretation) *An alternative interpretation of Equation 1 is that it approximates the influence of a training example on a test example* had it been visited *at each of the input checkpoints. Under this counterfactual interpretation, it is valid to study the influence of a point that is not part of the training data set, keeping in mind that such an example did not impact the training process or the checkpoints that arose as a consequence of the training process.*

4 Evaluations

In this section we compare `TracIn` with influence functions [2] and the representer point selection method [3]. Brief descriptions of these two methods can be found in the supplementary material. We also compare practical implementations of `TracIn` against an idealized version.

4.1 Evaluation Approach

We use an evaluation technique that has been used by the previous papers on the topic (see Section 4.1 [3] and Section 5.4 of [2]).[6] 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 `TracIn` 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. (In our evaluation, we know which examples are mislabelled because we introduced them. If this technique was used to find mislabelled examples in practice, we assume that a human would inspect the list and identify mislabelling.)

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.

4.2 CIFAR-10

In this section, we work with ResNet-56 [16] trained on the CIFAR-10 [17]. The model on the original dataset has 93.4% test accuracy. [7]

Identifying Mislabelled Examples Recall the evaluation set up and metric in Section 4.1.[8]

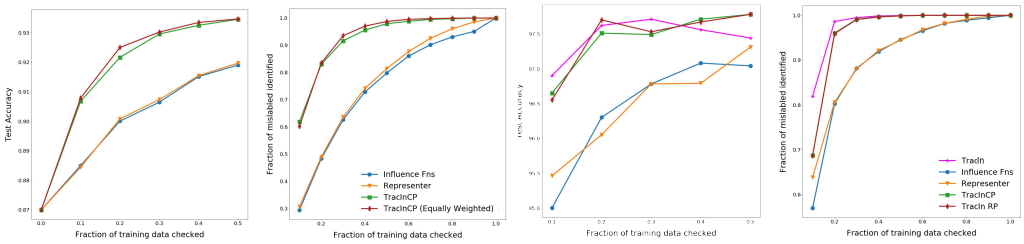
For influence functions, it is prohibitively expensive to compute the Hessian for the whole model, so we work with parameters in the last layer, essentially considering the layers below the last as frozen.

[5] In a sense, the checkpoint based approximation is an improvement on the idealized definition of `TracIn` because it ignores the order in which the training data points were visited by the specific training run; this sequence will change for a different run.

[6] This is just an evaluation approach. There are possibly other ways to detect mislabelled examples (e.g. [15]) that don't use the notion of training data influence/attribution.

[7] All model for CIFAR-10 are trained with 270 epochs with a batch size of 1000 and 0.1 as initial learning rate and the following schedule (1.0, 15), (0.1, 90), (0.01, 180), (0.001, 240) where we apply learning rate warm up in the first 15 epochs.

[8] Training on the mislabelled data reduces test accuracy from 93.4% to 87.0% (train accuracy is 99.6%).



(a) CIFAR-10 (b) MNIST

Figure 1: CIFAR-10 and MNIST Mislabelled Data Identification with `TracIn` Representer points, and Influence Functions. We use "Fraction of mislabelled identified" on the y axis to compare the effectiveness of each method. (RP = Random Projections, CP = CheckPoints)

This mimics the set up in Section 5.1 of [2]. 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 [3] to compare with self-influence.

We use `TracInCP` 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 `TracInCP` identifies a larger fraction of the mislabelled training data (y-axis) regardless of the fraction of the training data set that is examined (x-axis). For instance, `TracIn` 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 `TracIn` compared to the other methods (see the plot on the left hand side of Figure 1a). We also show that weighting checkpoints equally yields similar results. This provides support to ignore learning rate for implementation simplification.

Effect of different checkpoints on `TracIn` scores Next, we discuss the contributions of the different checkpoints to the scores produced by `TracIn`; recall that `TracIn` computes a weighted average across checkpoints (see the defintion of `TracInCP`). 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 Figure 8 (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.*

4.3 MNIST

In this section, we work on the MNIST digit classification task[9]. 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 3.2), and not just heuristic influence using checkpoints. Second, we can apply `TracIn` 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.

[9]We use a model with 3 hidden layers and 240K parameters. This model has 97.55% test accuracy.

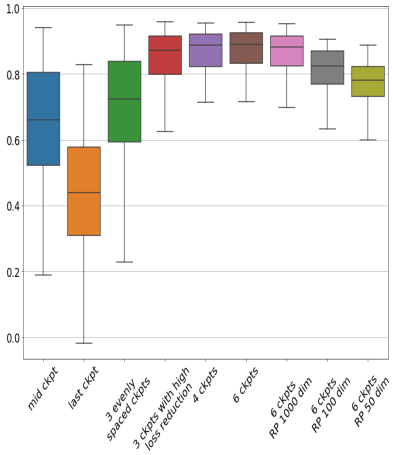
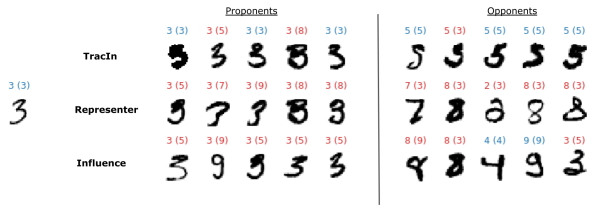
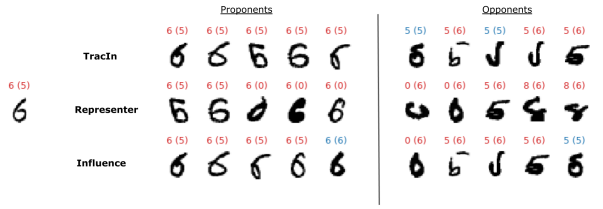


Figure 2: Analysis of effect of approximations with Pearson correlation of first order approximate `TracIn` influences with heuristic influences over multiple checkpoints and with projections of different sizes. RP stands for random projection.



(a) Correctly classified 3.



(b) Incorrectly classified 6

Figure 3: Proponents and opponents examples using `TracIn`, representer point, and influence functions. (Predicted class in brackets)

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 `TracInCP` 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 3a and '5' and '8's in Figure 3b). In contrast, `TracInCP` seems to pick pixel-wise similar opponents.

Identifying mislabelled examples. Recall the evaluation set up and metric in Section 4.1. We train on MNIST mislabelled data as described there.[10] Similar to CIFAR-10, `TracIn` 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, approximate `TracIn` is able to recover mislabelled examples faster than heuristic `TracInCP` (we use every 30th checkpoint, starting from 20th checkpoint), but not by a large margin.

Next, we evaluate the effects of our various approximations.[11]

Effect of the First-Order Approximation. We now evaluate the effect of the first-order approximation (described in 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 7 (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.

Effect of checkpoints. We now discuss the approximation from Section 3.3, i.e., the effect of using checkpoints. We compute the correlation of the influence scores of 100 test points using `TracInCP` with different checkpoints against the scores from the first-order approximation `TracIn`. As discussed in Remark D (in the supplementary material), 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 2 shows `TracInCP` with just one checkpoint from middle correlates more than the last checkpoint with

[10] After 140 epochs, it achieves accuracy of 89.94% on mislabelled train set, and 89.95% on test set.

[11] 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.

TracIn scores. Consequently, TracInCP with more checkpoints improves the correlation, more so if the checkpoints picked had high loss reduction rates.

4.4 Discussion

We now discuss conceptual differences between TracIn and the other two methods.

Influence functions: Influence functions mimic the process of tracing the change in the loss of a test point when you *drop* an individual training point and retrain. In contrast, as discussed in Section 3, TracIn explains the change in the loss of a test point between the start of training and the end of training. The former counterfactual is inferior when the training data set contains copies or near copies; deleting one of the copies is likely to have no effect even though the copies together are indeed influential. Also, it is prohibitively expensive drop a datapoint and retrain the model, influence functions approximate this by using the first and second-order optimality conditions. Modern deep learning models are rarely, if ever, trained to even moderate-precision convergence, and optimality can rarely be relied upon. In contrast, TracIn does not need to rely optimality conditions.

Representer Point method: This technique 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 ℓ_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. Like the influence function approach, it too relies on *optimality conditions*. Also, it can only explain the prediction of the test point, and not its loss.

Implementation Complexity: Both techniques are complex to implement. The influence technique requires the the inversion of a Hessian matrix that has a size that is quadratic in the number of model parameters, and the representer point method requires a complex, memory-intensive line search or the use of a second order solver such as LBFGS. TracIn, in contrast, has a simple implementation.

5 Applications

We apply TracIn to a regression problem (Section 5.1) a text problem (Section 5.2) and an computer vision problem (Section 5.3) to demonstrate its ability to generate insights. This section is not meant to be an evaluation. The last of these use cases is on a ResNet-50 model trained on the (large) Imagenet dataset, demonstrating that TracIn scales.

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 super stardom 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.

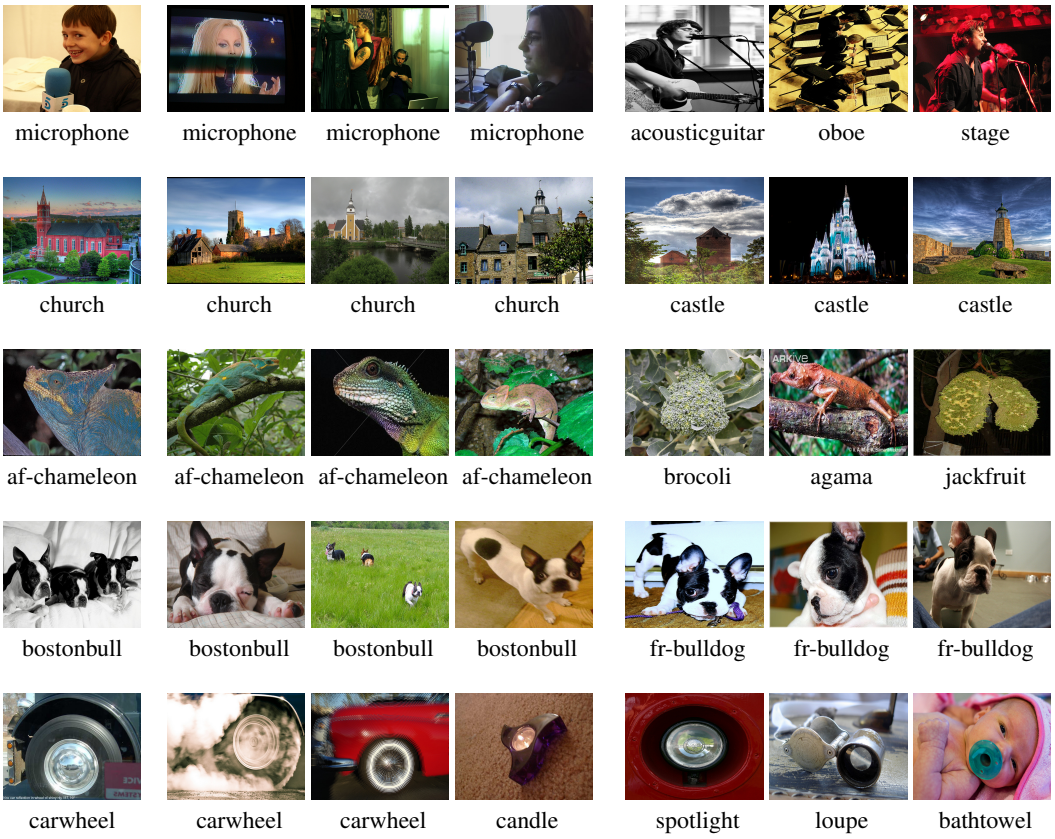


Figure 4: `TracIn` applied on Imagenet. Each row starts with the test example followed by three proponents and three opponents. The test image in the first row is classfied as band-aid and is the only misclassified example. (af-chameleon: african-chameleon, fr-bulldog: french-bulldog)

5.1 California Housing Prices

We study `TracIn` on a regression problem using California housing prices dataset [18]. [12]

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 `TracInCP` 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. 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 4.1 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. Housing plots with geo coordinates can be found in Figure 10 (in the supplementary material).

[12] We used a 8:2 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 `TracIn` influences.

5.2 Text Classification

We apply TracIn on the DBPedia ontology dataset introduced in [19]. 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) [20] for 60 epochs and use the default parameters of sentencepeice library as tokenizer [21] and achieve 95.5% on both training and test. We apply TracInCPand 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 disciminative.) The list of opponents provide insight about data introducing correlation between politicians and artists.

5.3 Imagenet Classification

We apply TracIn on the fully connected layer of ResNet-50 trained on Imagenet [22][13] We use a trick to reduce dimensionality: It 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, TracIn involves computing the dot (Hadamard) product of two rank 1 matrices. Details are in the supplementary material.

We show three proponents and three opponents for five examples in figure 4. 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 4, 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.

6 Conclusion

In this paper we propose a method called TracIn to identify the influence of a training data point on a test point.

The method is *simple*, a feature that distinguishes it from previously proposed methods. Implementing TracIn only requires only a basic understanding of standard machine learning concepts like gradients, checkpoints and loss functions.

The method is *general*. It applies to any machine learning model trained using stochastic gradient descent or a variant of it, agnostic of architecture, domain and task.

The method is *versatile*. The notion of influence can be used to explain a single prediction (see Section 5), or identify mislabelled examples (see Section 4). Over time, we expect other applications to emerge. For instance, [23] uses it to perform a kind of active learning, i.e., to expand a handful of hard examples into a larger set of hard examples to fortify toxic speech classifiers.

Finally, we note that some human judgment is required to apply TracIn correctly. To feed it reasonable inputs, i.e., checkpoints, layers, and loss heads. To interpret the output correctly; for instance, to inspect sufficiently many influential examples so as to account for the loss on the test example. Lastly, as with any statistical measure (e.g. mutual information Pearson correlation etc.), we need to ensure that the measure is meaningfully utilized within a broader context.

7 Acknowledgements

We would like to thank the anonymous reviewers, Qiqi Yan, Binbin Xiong, and Salem Haykal for discussions.

[13] The model is trained for 90 epochs and achieves 73% top-1 accuracy. The training data consists of 1.28M training examples with 1000 classes. The 30th, 60th, and 90th checkpoints are used for TracInCP and we project the gradients to a vector of size 1472.

8 Broader Impact

This paper proposes a practical technique to understand the influence of training data points on predictions. For most real world applications, the impact of improving the quality of training data is simply to improve the quality of the model. In this sense, we expect the broader impact to be positive.

For models that impact humans, for instance a loan application model, the technique could be used to examine the connection between biased training data and biased predictions. Again, we expect the societal impact to be generally positive. However there is an odd chance that an inaccuracy in our method results in calling a model fair when it is not, or unfair, when it is actually fair. This potential negative impact is amplified in the hands of an adversary looking to prove a point, one way or the other. It is also possible that the technique could be used to identify modifications to the training data that hurt predictions broadly or for some narrow category.

References

[1] TracIn Code. `https://github.com/frederick0329/TracIn`.

[2] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1885–1894, 2017.

[3] Chih-Kuan Yeh, Joon Kim, Ian En-Hsu Yen, and Pradeep K Ravikumar. Representer point selection for explaining deep neural networks. In *Advances in Neural Information Processing Systems*, pages 9291–9301, 2018.

[4] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3319–3328, 2017.

[5] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*, pages 4765–4774, 2017.

[6] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " 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*, pages 1135–1144, 2016.

[7] Art B Owen and Clémentine Prieur. On shapley value for measuring importance of dependent inputs. *SIAM/ASA Journal on Uncertainty Quantification*, 5(1):986–1002, 2017.

[8] Art B Owen. Sobol'indices and shapley value. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):245–251, 2014.

[9] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nick Hynes, Nezihe Merve Gürel, Bo Li, Ce Zhang, Dawn Song, and Costas J Spanos. Towards efficient data valuation based on the shapley value. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1167–1176, 2019.

[10] Amirata Ghorbani and James Zou. Data shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning*, pages 2242–2251, 2019.

[11] Satoshi Hara, Atsushi Nitanda, and Takanori Maehara. Data cleansing for models trained with sgd. In *Advances in Neural Information Processing Systems*, pages 4215–4224, 2019.

[12] Rico Sennrich, Barry Haddow, and Alexandra Birch. Edinburgh neural machine translation systems for WMT 16. In *Proceedings of the First Conference on Machine Translation*, pages 371–376, 2016.

[13] Samuli Laine and Timo Aila. Temporal ensembling for semi-supervised learning. In *5th International Conference on Learning Representations*, 2017.

[14] Hao-Ran Wei, Shujian Huang, Ran Wang, Xin-Yu Dai, and Jiajun Chen. Online distilling from checkpoints for neural machine translation. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 17th Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1932–1941, 2019.

[15] Yanyao Shen and Sujay Sanghavi. Learning with bad training data via iterative trimmed loss minimization. In *ICML*, 2019.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, Toronto, Canada, 2009.

[18] R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

[19] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657, 2015.

[20] Dinghan Shen, Guoyin Wang, Wenlin Wang, Martin Renqiang Min, Qinliang Su, Yizhe Zhang, Chunyuan Li, Ricardo Henao, and Lawrence Carin. 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)*, pages 440–450, 2018.

[21] Taku Kudo and John Richardson. 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*, pages 66–71, 2018.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[23] Xiaochuang Han and Yulia Tsvetkov. Fortifying toxic speech detectors against veiled toxicity, 2020.

[24] R Dennis Cook and Sanford Weisberg. *Residuals and influence in regression*. New York: Chapman and Hall, 1982.

[25] Barak A. Pearlmutter. Fast exact multiplication by the hessian. *Neural Computation*, 6:147–160, 1994.

[26] Bernhard Schölkopf, Ralf Herbrich, and Alex J Smola. A generalized representer theorem. In *International conference on computational learning theory*, pages 416–426. Springer, 2001.

[27] Erik Bernhardsson. *Annoy: Approximate Nearest Neighbors in C++/Python*, 2018. Python package version 1.13.0.

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

A Description of Influence Functions and Representer Point Methods

A.1 Influence Functions

[2] proposed using the idea of Influence functions [24] 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:

$$\text{Inf}(z, z') = -\nabla_{\hat{w}}\ell(\hat{w}, z') \cdot H_{\hat{w}}^{-1} \cdot \nabla_{\hat{w}}\ell(\hat{w}, z). \tag{2}$$

Here, $H_{\hat{w}} = \frac{1}{n}\sum_1^n \nabla^2\ell(\hat{w}, z_i)$ is the Hessian. As pointed out by [2], 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.

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 E. 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 $\min_S \|H_{\hat{w}}S - G^\top\|_F^2$, via a customized stochastic gradient descent procedure based on the formula

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

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

$$2[\tfrac{1}{|B_1|}\textstyle\sum_{z\in B_1}\nabla^2\ell(\hat{w}, z)][\tfrac{1}{|B_2|}\sum_{z\in B_2}\nabla^2\ell(\hat{w}, z)S - G^\top]. \tag{3}$$

Note that $\mathbb{E}[\frac{1}{B_1}\sum_{z\in B_1}\nabla^2\ell(\hat{w}, z)] = H_{\hat{w}}$ and $\mathbb{E}[\frac{1}{|B_2|}\sum_{z\in B_2}\nabla^2\ell(\hat{w}, z)] = H_{\hat{w}}$, and since B_1 and B_2 are independently chosen, we conclude that the expectation of the quantity in (3) is indeed $2H_{\hat{w}}(H_{\hat{w}}S - G^\top)$ as required. Note that (3) can be computed using Hessian-vector products, which can be computed easily using the Pearlmutter trick [25].

A.2 Representer Point Selection

The second method is proposed in [3] and is based on the representer point theorem [26]. 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

$$\begin{aligned}\text{Rep}(z, z') = \\ -\frac{1}{2\lambda n}(f(w_1, z)\cdot f(w_1, z'))\partial_{\phi(w', z')}\ell(w', z').\end{aligned} \tag{4}$$

Here, $f(w_1, z)$ is the input representation, i.e. the outputs of the last hidden layer, and $\phi(w', z) = w_2' f(w_1, z)$ 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

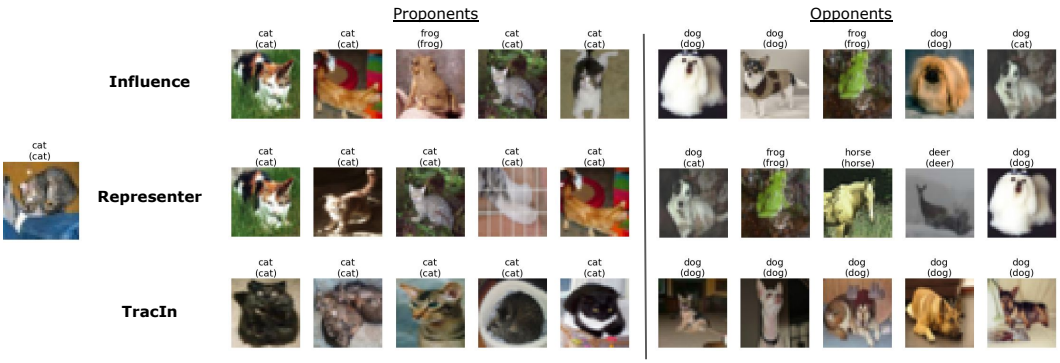


Figure 5: CIFAR-10 results: Proponents and opponents examples of a correctly classified cat for influence functions, representer point, and `TracIn`. (Predicted class in brackets)

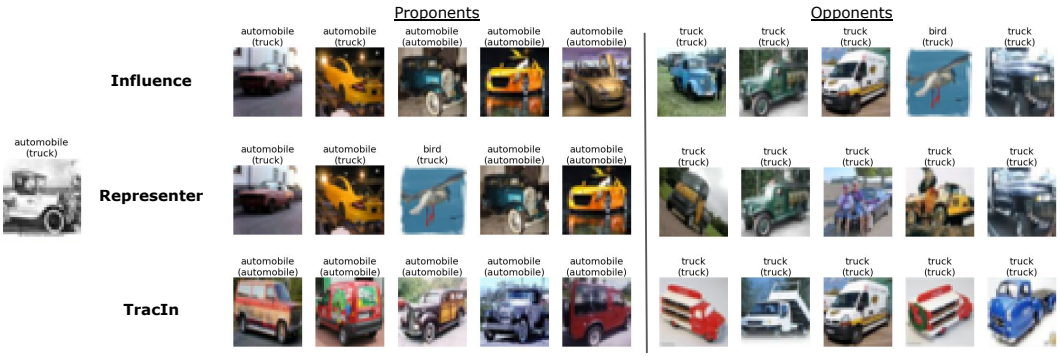


Figure 6: CIFAR-10 results: Proponents and opponents examples of an incorrectly classified automobile for influence functions, representer point, and `TracIn`. (Predicted class in brackets)

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 `TracIn` 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 5 and 6. All three methods retrieved mostly cats as positive examples and dogs as negative examples, but `TracIn` seems more consistent on the types of cats and dogs. For the mis-classified automobile, proponents of `TracIn` pick up automobiles of a similar variety type.

C 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, see Section E in appendix for details). Then, we concatenate the loss gradients for a given training point z (i.e., $\ell(w_{t_1}, z), \ell(w_{t_2}, z) \ldots \ell(w_{t_k}, z)$) together into one vector. This can be then loaded into an approximate nearest neighbor library(e.g. [27]). 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 `TracInCP`.

D Selecting Checkpoints

In the application of `TracInCP`, 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 `TracInCP` with *just* the final model could result in noisy results.

E 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 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 [28]) 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 $\mathbb{E}[G^\top G] = I$. We compute the following sketch: in iteration t, compute and save $\eta_t G \nabla \ell(w_t, z_t)$. Then given a test point z', the dot product $(\eta_t G \nabla \ell(w_t, z_t)) \cdot (G \nabla \ell(w_t, z'))$ is an unbiased estimator of $\eta_t \nabla \ell(w_t, z_t)) \cdot (\nabla \ell(w_t, z'))$, and can thus be substituted in all influence computations.

F 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 the n is the number of units in the output of the layer. For the purpose of `TracIn` 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:

$$\nabla_W (f(Wx)) = \nabla_y f(y) x^\top.$$

In particular, note that the gradient w.r.t. W is rank 1. This property is very useful for `TracIn` 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

$$\begin{aligned}
&\nabla_W (f(Wx)) \cdot \nabla_W (f'(Wx')) \\
&= (\nabla_y f(y) x^\top) \cdot (\nabla_{y'} f'(y') {x'}^\top) \\
&= (\nabla_y f(y) \cdot \nabla_{y'} f'(y'))(x \cdot x').
\end{aligned}$$

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 `TracIn` 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 $\mathbb{E}[G_1 G_1^\top] = \mathbb{E}[G_2 G_2^\top] = I$, 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

$$\begin{aligned}&\mathbb{E}[(G_1 \nabla_y f(y) x^\top G_2^\top) \cdot (G_1 \nabla_{y'} f'(y') {x'}^\top G_2^\top)] \\ &= (\nabla_y f(y) x^\top) \cdot (\nabla_{y'} f'(y') {x'}^\top),\end{aligned}$$

so the randomized dot-product is unbiased.

G Additional Results

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

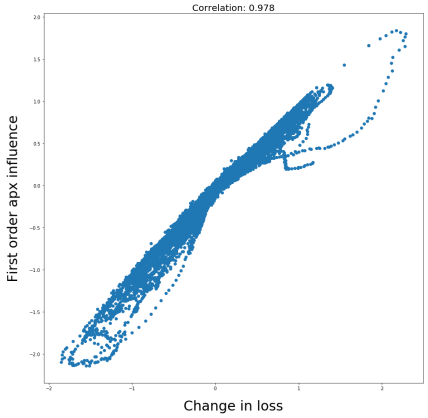


Figure 7: Comparison of change in loss at all training steps and `TracIn` influences at those steps for 100 test examples from MNIST dataset. This measures the quality of the first-order approximation–see Section 4.3.

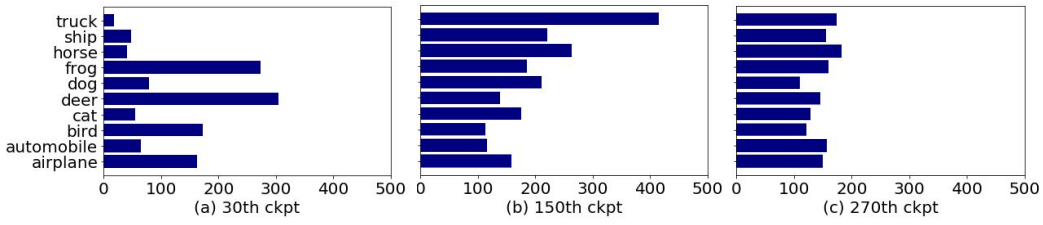
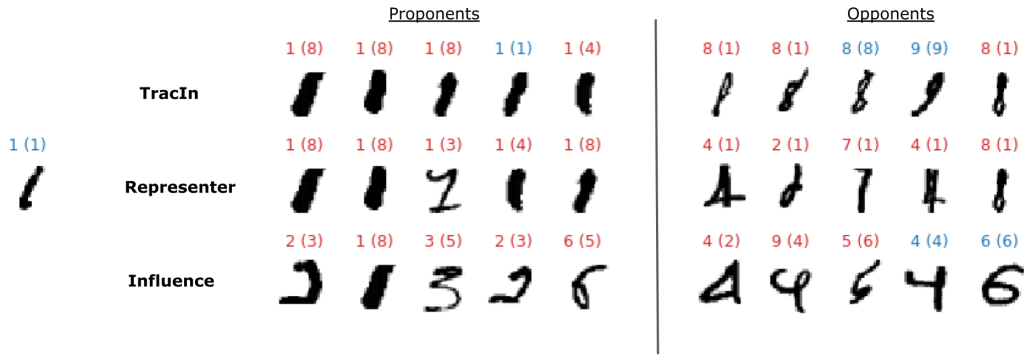
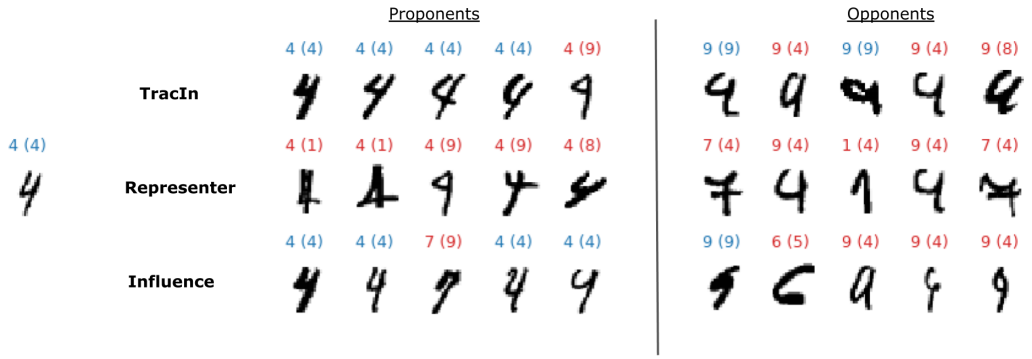


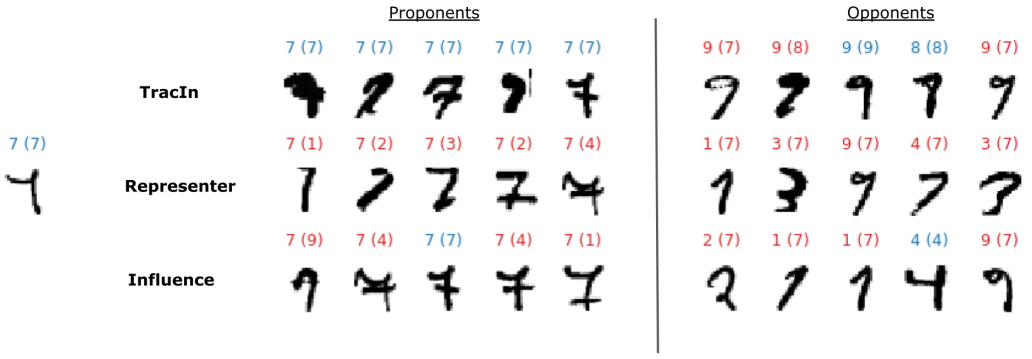
Figure 8: Number of identified mislabelled examples by class for three checkpoints within the top 10% of ranking by self-influence. Different checkpoints highlight different labels—see Section 4.2.



(a)

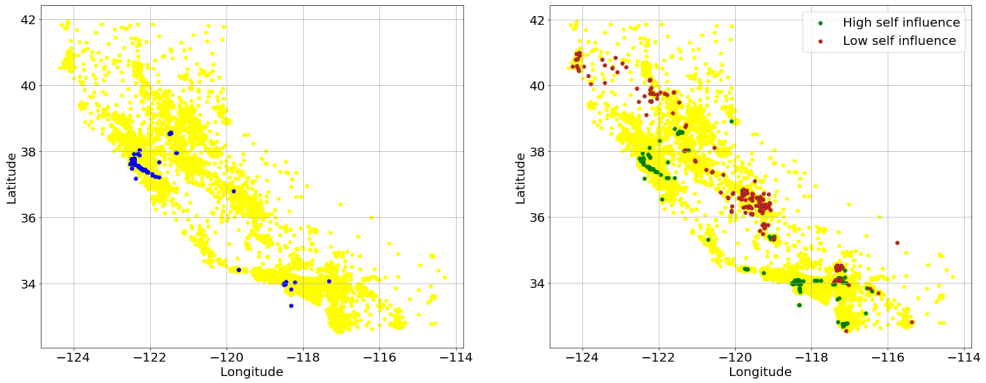


(b)



(c)

Figure 9: MNIST(Section 4.3): Proponents and opponents examples of a correctly classified images for `TracIn`, representer point, and influence functions. (Predicted class in brackets).



(a) Influential training examples for 11 test examples in city of Palo Alto, with entire dataset in yellow.

(b) Training examples with high and low self influences showing dense areas which model memorizes, and sparsely populated areas where model learns from examples away from the area.

Figure 10: `TracIn` on California housing prices dataset.

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	**José Manuel Restrepo Vélez** José Manuel Restrepo Vélez (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: 陳家强) 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.