# Leveraging Small Software Engineering Data Sets with Pre-trained Neural Networks

Romain Robbes
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

Andrea Janes
Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

*Abstract*—**Many software engineering data sets, particularly those that demand manual labelling for classification, are necessarily small. As a consequence, several recent software engineering papers have cast doubt on the effectiveness of deep neural networks for classification tasks, when applied to these data sets. We provide initial evidence that recent advances in Natural Language Processing, that allow neural networks to leverage large amount of unlabelled data in a pre-training phase, can significantly improve performance.**

*Index Terms*—**Data Sets, Deep Learning, Transfer Learning**

## I. Introduction

Deep neural networks have revolutionized the field of machine learning, leading to significant improvements over the state of the art in a large variety of tasks and domains (including image recognition, speech recognition, machine translation, summarization, and others). In Software Engineering, the results have been more mixed. While many approaches have promising results (e.g., code completion, variable naming, code translation and summarization), several recent papers have cast doubt about the effectiveness of neural networks in some settings [1], [2]. One reason is the performance on small SE data sets, which we examine in this paper.

The use cases where deep neural networks outperform the state of the art are those where large amount of data is available, such as the ImageNet data set [3], which features more than 14 million labelled images in its largest version. However, many data sets are much smaller, particularly in the case of supervised learning, as each data point must be manually labelled. This manual task is often an expensive process, and all the more in Software Engineering: while ImageNet labelling can be crowdsourced, labelling a SE data set requires significant expertise. It cannot be crowdsourced easily, and is thus vastly more expensive. In practice, many SE data sets are manually labelled by researchers, and are thus limited to hundreds or thousands of data points only. Recent work reports disappointing performance when training deep neural networks from scratch on small SE data sets.

In this paper, we investigate whether the small data set issue can be alleviated in Software Engineering, starting with natural language (NL) SE datasets (we are actively exploring the source code case). We show that a form of transfer learning, namely unsupervised pre-training on large data sets, can be effectively used to improve the performance on small NL SE data sets. Several approaches—detailed in Section III—have shown this strategy to be effective in Computer Vision and (lately) in Natural Language Processing. These approaches are all a variation of a common idea: when training a neural network from scratch, all the model's weights are initialized randomly. Pre-training instead initializes the weights with values obtained by training a model on a related tasks, for which data is easier to gather; in this case, by using large amounts of unlabelled data, no manual labelling is necessary.

## II. Small Data Sets in Software Engineering

Many labelling tasks, such as identifying objects in images, drawing bounding boxes for these objects, or identifying the sentiment of a movie review, do not require specific expertise. They can hence be crowdsourced at a reasonable cost, particularly thanks to crowdsourcing platforms such as Amazon Mechanical Turk.

As manual labelling in Software Engineering requires expertise in Software Engineering, it is usually done by the researchers themselves. This approach is expensive, and due to the amount of effort involved, is often limited to hundreds or thousands of data points. This is all the more true since the effort is often duplicated to verify whether annotators are in agreement with each other. Some examples follow.

Bacchelli et al. manually linked 2,139 emails from 6 open-source projects with the source code entities they referenced [4]. Fakhoury et al. labelled 1,700 Java methods (with comments) according to the linguistic antipatterns they exhibited [2]. Ortu et al. labelled 2,000 JIRA issue reports, and 4,000 sentences with the sentiments expressed in them. Novielli et al. labelled 4,423 stack Overflow posts with sentiment [5]. Lin et al. labelled the sentiment of *each word* of 1,500 Stack Overflow sentences according to their sentiment [1]; they also labelled 341 app reviews for sentiment. Villaroel et al. [6] classified 1,200 app reviews, while Maalej at al. [7] classified 4,400 app reviews, in both cases in categories such as bug reports, and feature requests. Zhou et al. hired students to analyze 1,674 API elements to determine whether they have API directive defects [8]. These data sets are several orders of magnitude smaller than the ones where Deep Learning approaches are the state of the art, such as ImageNet.

In two cases, some of these data sets have been used as input for deep learning techniques, with lackluster results. Fakhoury et al. applied a Convolutional Neural Network (CNN) for linguistic antipattern detection; they find that the CNN matches the performance of an SVM, but that parameter tuning of the SVM further increases performance [2]; in contrast, tuning the hyperparameters of the CNN is impractical as training it is very slow. Lin et al. train the Stanford CoreNLP parser, that uses an RNN, on Stack Overflow, JIRA, and App Review data for sentiment analysis, and finds that its performance is below their expectation; in fact, they find that all of the tools that perform sentiment analysis that they tried, are generally unsuitable for the Software Engineering domain [1].

## III. Pre-training Neural Networks

Neural network pre-training is a form of inductive transfer learning [9], in which a neural network is first trained on a source task (supervised or unsupervised), before some, or all, of the layers of the neural network are trained on a target task. The training on the source task is often called pre-training, while the training on the target task is often called fine-tuning.

The intuition behind this practice is that the weights learned during pre-training are a much better "starting point" than randomly initialized weights.

*a) Pre-training in Computer Vision:* Pre-training is a very widespread practice in computer vision. Early unsupervised pre-training algorithms via autoencoders made training of deep neural networks easier [10], [11]. With the advent of the ImageNet data set [3], the practice of supervised pre-training on ImageNet [12] emerged. In this case, a neural network is trained to classify a version of the ImageNet data set (1,300,000+ images, divided in 1,000 classes), and is then re-purposed to perform similar tasks (such as different classification, image segmentation, or others [12]). The last classification layers of the network are replaced either by new classification layers, or layers performing a different task. Training on the new task is then resumed to learn weights for the new layers, and adjust the weights of the first layers.

Perhaps an intuitive explanation of why pre-training works comes from visualizing the layers of a CNN [13]. Zeiler and Fergus show that the layers of a CNN learn more and more abstract features. Taking the example of a neural network with five layers trained on ImageNet, they show that the neurons in each layer are activated by different types of features, with increasing levels abstraction. Neurons in the first layer learn to detect edges or patches of color; the second layer detects simple patterns such as corners, circles, or parallel lines; the third layer more intricate patterns (grids, text); the fourth layer is more class-specific (e.g. detecting dog faces), while neurons in the fifth layer tend to be activated by entire objects, with diversity in their configuration (e.g. detecting various types of dogs). Clearly, many of the features learned in such a network are rather general, and can be reused for other detection tasks.

A further practical consequence of this is that after training a model, researchers can provide it to the community. Other researchers can then download the model weights to reuse them in their specific scenarios. The practice has become so commonplace that some deep learning APIs allow extremely easy access to these models; instantiating a pre-trained model is just a (large) download and a few lines of code away.

*b) Pre-training in Natural Language Processing:* Comparatively, pre-training has been, until recently, less prevalent in Natural Language Processing. Beyond being an area where sustained interest in applying deep neural networks is more recent, another reason is that NLP data exhibits more variability. While it is reasonable to build a data set where all the images have the same dimensions, sentences and documents can vary greatly in length. In addition, at the most common unit of modelling—when documents are modelled as sequences of words—an additional challenge is how to handle words that are unknown to the model (the "out of vocabulary" problem).

The first advances in this domain has been word-embeddings, particularly with word2vec [14]. These approaches learn word representations (word vectors) on an unlabelled corpus based on co-occurrences of words in a similar context (usually a window of 2 words before/after the word of interest). Pre-trained word-embeddings are easily available and are commonly used in Software Engineering, and researchers have provided word2vec embeddings specific to Software Engineering [15].

While a step in the right direction, word embeddings are far from a full pre-trained deep model: they are shallow. Word embeddings are essentially the first layer of a neural network dedicated to NLP.

In the last year, several approaches showed that it was possible to apply unsupervised pre-training of deeper models in NLP, showing significant improvements on the target tasks. Howard and Ruder present a specific training schedule enabling pre-training of full LSTM [16]; we adopt their approach, ULMFit, described next. During this study, additional work was published: Peters et al. introduced Embeddings from Language Models (ELMo), where more descriptive embeddings are created based on the internal state of an LSTM [17]. Radford et al. show that they can pre-train a different neural network architecture, the Transformer [18].

## IV. ULMFit

In this work, we apply the Universal Language Model Fine-tuning (ULMFit) approach of Howard and Ruder [16], and test whether the improvements they observed on text classification can be obtained on Software Engineering data sets, including on language models of source code. We first start by describing the approach in details. ULMFit consists in three steps: 1) a Language Model pre-training step, where a language model is trained (unsupervised) on a general domain (source) corpus; 2) a Language Model fine-tuning step, where the language model is fine-tuned (still unsupervised) on the target corpus; and 3) a classifier fine-tuning step, where the final classifier is fine-tuned in a supervised manner.

During language model pre-training, the language model is trained normally. That is, a Long short-term memory (LSTM) [19] language model (more precisely, the AWD-LSTM [20]) is trained to predict the next word in the corpus, given the previous words in a sequence. Howard and Ruder reuse an existing pre-trained language model, trained on the Wikitext-

103 data set by Merity et. al [20]; this corpus contains 28,595 Wikipedia articles, for a total of 103 million words.

In the second step, language model fine-tuning, the language model is fine-tuned to the specific characteristics of the target corpus. In particular, the target corpus is scanned for unknown words (words that were absent from the source corpus). The new words are added to the model's vocabulary: the embedding matrix is expanded to accommodate them, and their weights are initialized as the mean of the weights of the other words. Afterwards, the model processes the target corpus to adjust the weights of the model to the new corpus. The fine-tuning approaches uses *discriminative fine-tuning*, that is, a different learning rate for each layer of the neural network. The rationale for this is that different layers model different kinds of information: the first layers, being more general, should adapt more slowly to new data than the last layers. The approach also varies the learning rate over time according to a schedule. This particular schedule uses *slanted triangular learning rates*, where the learning rate initially starts slow, rises quickly, before dropping gradually.

Finally, in the third step, the language model is re-purposed for classification: the softmax layer is replaced by two linear blocks with batch normalization and dropout. As these layers are initialized with random weights, simply resuming training would cause too many updates in the earlier layers, risking forgetting valuable knowledge. Thus, the model is gradually unfreezed: during the first training cycle, all but the last layers are frozen, meaning that their weights will not be updated during training. In the next cycle, the next to last layer is unfrozen before training resumes; the process continues until all layers are unfrozen. This fine-tuning also uses discriminative fine tuning and slanted triangular learning rates.

## V. Sentiment Analysis: Going Farther

To test whether pre-training works in the Software Engineering domain, we applied the ULMFit approach to the Sentiment Analysis case, using the three data sets from the study by Lin et al. [1]: Stack Overflow sentences (1,500 sentences), App Reviews (341 app reviews), and JIRA issues (926 issues). We trained the AWD-LSTM [20] model both from scratch, and using the ULMFit pre-training approach. In the latter case, similarly to Howard and Ruder, we used the pre-trained Wikitext-103 model by Merity. This model has three LSTM layers with 1,150 hidden units, and uses an embedding size of 400. We perform 5-fold cross validation (training on 80% of the data, and validating on 20%). We report the average performance over all splits.

We found that the size of the data sets were small enough that hyper-parameter search was feasible for the fine-tuning phase (training a model using 5-fold cross validation varies, but takes between ca. 5-15 minutes). We performed random hyper-parameter search, varying the most important hyper-parameters: learning rate, learning rate multiplier, dropout, and weight decay. Other parameters used the values reported by Howard and Ruder. For each data set, we tried 275 to 330 random configurations, and report the performance of the best performing one.

Similarly to Lin et al., as a first measure of performance we report the accuracy of the best-performing models (both from scratch and pre-trained) and compare with the accuracy of all the models that were reported in Lin et al.: Stanford CoreNLP SO, Stanford CoreNLP, SentiStrength-SE, SentiStrength, and NLTK. We also report the F-score (harmonic mean of precision and recall) for all the classes; Lin et al. reported precision and recall independently. We prefer the F-score since it consolidates precision and recall in a single metric, and favours models that have a balanced performance (a model with, eg, very low precision and very high recall will have a low f-score). We report the performance of all approaches on all data sets in table I, as well as the average of the accuracy across all three data sets.

We can see that across all the data sets, both AWD-LSTM approaches perform better and more consistently than the other approaches in terms of accuracy. This is not surprising, since they are the only approaches that are trained on the data itself, with the exception of Stanford Core NLP SO, which is trained on Stack Overflow (more on this later). As Lin et al. observed, off the shelf techniques are not ready for use yet, and specific re-training is required. Specific re-training is indeed the only way to obtain consistently acceptable performance, which makes any approach that improves performance for small data sets valuable.

In this context, the most important observation is that the pre-trained AWD-LSTM indeed outperforms the one trained from scratch, and especially does so on the data sets with the lowest accuracy. Over all datasets, the pre-trained version improves accuracy by 2.9%, which corresponds to a 21.4% reduction in error rate. Moreover, the pre-trained AWD-LSTM is the only approach that has reasonable (exceeding 50%) F-scores for the positive and negative classes on the Stack Overflow data set. The only issue we can observe is the neutral class on the App Reviews data set, which the pre-trained AWD-LSTM essentially ignores. This is because there are only 25 instances of that class in the entire data set, which is extremely small, especially with 5-fold cross validation (other configurations did fare better for this class, but never well).

Since Stanford CoreNLP SO and the AWD-LSTMs are trained on the Stack Overflow data set, we can perform a more head-to-head comparison. We can see that in this case, our approaches have better performance, in spite of two limitations: we use five-fold cross validation instead of ten-fold cross validation, so we train on only 80% of the data, not 90%. More importantly, our approach only requires one label per sentence, not one label per word. This is a key difference, as this allows our approach to be re-trained much more easily on data that is labelled at the sentence level, such as the App Review and the JIRA data set. Since per-word labels were lacking on these two data sets, Lin et al. were unable to retrain their classifier without investing significant manpower to do so.

TABLE I
OUR RESULTS (SHADED) COMPARED TO LIN ET AL. [1] (WHITE BACKGROUND). BOLD INDICATES BEST PERFORMANCE; UNDERLINED, SECOND-BEST.

| | Stack Overflow | | | | | App Reviews | | | | | JIRA Issues | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tool | Correct Predictions | Accuracy | Positive F-score | Neutral F-score | Negative F-score | Correct Predictions | Accuracy | Positive F-score | Neutral F-score | Negative F-score | Correct Predictions | Accuracy | Positive F-score | Neutral F-score | Negative F-score | Average accuracy |
| SentiStrength | 1,043 | 69.5 | 25.7 | 81.3 | 41.4 | 213 | 62.5 | 80.1 | 16.7 | 47.8 | 714 | 77.1 | 88.4 | N/A | 82.3 | 69.7 |
| NLTK | 1,168 | 77.9 | 27.6 | 87.3 | 14.8 | 184 | 54.0 | 78.0 | 15.4 | 28.9 | 276 | 29.8 | 50.6 | N/A | 42.4 | 53.9 |
| Stanford CoreNLP | 604 | 40.3 | 27.6 | 49.5 | 29.2 | 237 | 69.5 | 76.9 | **20.3** | 70.8 | 626 | 67.6 | 66.9 | N/A | 80.5 | 59.1 |
| SentiStrength-SE | 1,170 | 78.0 | 25.9 | 87.5 | 27.0 | 201 | 58.9 | 77.7 | 16.8 | 45.4 | 704 | 76.0 | 91.4 | N/A | 82.5 | 71.0 |
| Stanford CoreNLP SO | 1,139 | 75.9 | 19.9 | 86.0 | 36.5 | 142 | 41.6 | 38.1 | 13.3 | 55.2 | 333 | 36.0 | 36.1 | N/A | 52.3 | 51.2 |
| AWD-LSTM (from scratch) | 1,225 | 81.7 | 23.0 | 90.0 | 39.4 | 272 | 79.8 | 85.5 | 7.1 | 78.9 | 880 | 95.0 | 91.9 | N/A | 96.4 | 85.5 |
| AWD-LSTM (pre-trained) | **1,273** | **84.9** | **53.5** | **91.3** | **58.2** | **288** | **84.5** | **90.5** | 0.0 | **83.7** | **894** | **96.5** | **94.4** | N/A | **97.5** | **88.6** |

## VI. CONCLUSIONS AND FUTURE WORK

Labelled SE data sets are necessarily small, as labelling them is expensive; any approach able to improve performance in these conditions can be extremely valuable. We investigated the effectiveness of unsupervised pre-training of neural networks on a large unlabelled corpora, followed by fine-tuning on the target task. While we have shown early evidence that unsupervised pre-training is beneficial for small NL SE data sets, extensive future work is needed.

*Unlabelled corpus.* We used a model pre-trained on Wikipedia but, "Wikipedia English" is quite different from the English on Stack Overflow, JIRA, or App Reviews. There, the language can be much more technical, or more informal than on Wikipedia. Typos may be much more common too. We expect that pre-training on unlabelled corpora of Stack Overflow posts, JIRA issues, or App Reviews would yield further performance improvement, and plan to quantify it.

*Other architectures.* In parallel to this work, novel architectures supporting pre-training emerged, such as ELMo embeddings [17] and the Transformer [18]. They could also be evaluated on small SE data sets as well. In addition, we only tried a forward LSTM, while Howard and Ruder report better performance with a bidirectional LSTM.

*New data sets*. We will conduct similar studies on additional NL SE data sets to see whether the effect is observable beyond sentiment analysis, and better characterize the performance difference when the size of the data set changes. Some additional data sets may require adaptations of the technique.

*Source code*. Source code data sets require extensive further work. For instance, Hellendoorn and Devanbu observed that the vocabulary in source code can grow much larger than in NL, as programmers are free to create arbitrarily long and complex identifiers [21]; careful modelling choices (such as identifier splitting) are needed to address these issues in order to maintain vocabulary size under control, and to handle project-specific identifiers.

*Providing models*. Finally, as we will be building pre-trained models based on specific corpora, we plan to release those pre-trained models to the community, so that other researchers can reuse them in their work, without incurring the prohibitive cost of training very large models from scratch.

## ACKNOWLEDGEMENTS

Results computed on the Vienna Scientific Cluster (VSC).

## REFERENCES

[1] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?" in *Proceedings of ICSE 2018*.

[2] S. Fakhoury, V. Arnaoudova, C. Noiseux, F. Khomh, and G. Antoniol, "Keep it simple: Is deep learning good for linguistic smell detection?" in *Proceedings of SANER 2018*.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of CVPR 2009*.

[4] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of ICSE 2010*.

[5] N. Novielli, F. Calefato, and F. Lanubile, "A gold standard for emotion annotation in stack overflow," *arXiv preprint arXiv:1803.02300*, 2018.

[6] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of ICSE 2016*.

[7] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requir. Eng.*, vol. 21, no. 3, 2016.

[8] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *Proceedings of ICSE 2017*.

[9] S. J. Pan, Q. Yang *et al.*, "A survey on transfer learning," *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, 2010.

[10] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, 2006.

[11] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Proceedings of NIPS 2007*.

[12] M. Huh, P. Agrawal, and A. A. Efros, "What makes imagenet good for transfer learning?" *arXiv preprint arXiv:1608.08614*, 2016.

[13] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proceedings of ECCV 2014*.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of NIPS 2013*.

[15] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proceedings of MSR 2018*.

[16] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," in *Proceedings of ACL 2018*.

[17] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of NAACL 2018*.

[18] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training."

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, 1997.

[20] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing LSTM language models," in *Proceedings of ICLR 2018*.

[21] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017.