Crafting Adversarial Attacks on Recurrent Neural Networks

Mark Anderson, Andrew Bartolo, and Pulkit Tandon
{mark01, bartolo, tpulkit}@stanford.edu

***Abstract*— We developed adversarial input generators to attack a recurrent neural network (RNN) used to classify the sentiment of IMDb movie reviews as being positive or negative. To this end, we developed LSTM network as well as two baseline models - SVM and Naïve Bayes and evaluated their accuracy under the attack by two black-box adversaries and a white box adversary. Our results showed that though LSTM is more robust than other two models, it's still very susceptible to white-box attack with generated adversary still preserving the sentiment of input review, making us question whether LSTMs really learn the sentiment in this task.**

I. INTRODUCTION

Today, RNNs are used in many applications, ranging from natural language processing to financial trading decision making. In particular, LSTMs [1] are highly effective for classifying sequential or time-series data. However, like convolutional neural nets, RNNs are vulnerable to adversarial examples. For instance, a cleverly-placed word may change the predicted sentiment of a movie review from positive to negative. Similar adversarial techniques can be applied to more safety-critical applications.

Our work assesses the vulnerability of classification-oriented (as opposed to sequence-oriented) LSTMs to three distinct types of adversarial attacks. Two of these attacks rely upon Naïve Bayes-based probabilities to generate adversarial examples, and the third relies upon a new gradient-based method, the Jacobian Saliency Map Approach (JSMA) [15]. For comparison, we also implemented two baseline classifiers — multinomial Naïve Bayes and a linear SVM — and measured their vulnerability to the adversarial attacks as well.

All three models are binary classifiers and make use of the same dataset, the Stanford Large Movie Review Database [2]. The goal of each model is to classify examples in the test set as accurately as possible. The goal of each adversary is to perturb the testing examples such that the model misclassifies the perturbed example as belonging to the opposite category.

II. RELATED WORK

Many papers about adversarial machine learning have focused on convolutional neural nets (CNNs) as benchmarked in [4], but relatively few have considered RNNs/LSTMs. Though some similarities exist between attacks on CNNs and RNNs, RNNs' discrete and sequential nature poses added challenges in generating and interpreting adversarial examples. To generate adversarial examples, the main idea is to find *close* inputs that have a high value of the loss function. One common way to find such inputs in image classifiers is to change each pixel, based on the gradient of the loss function near the input. This basically considers a linear approximation of the loss function around the inputs. Due to the discrete nature of RNNs, such an approach doesn't directly work. However, we can use the same intuition to find discrete modifications to the inputs that roughly align with the gradient of the loss function. [15] showed that the Jacobian Saliency Map Approach, though initially developed for feed-forward networks, can be generalized to RNNs. [16] extended this approach to generate adversarial examples using Generative Adversarial Networks (GANs).

III. DATASET AND FEATURES

All models and adversaries in this project operated on the same dataset, the Stanford Large Movie Review Dataset [2]. This dataset consists of 50,000 reviews of movies gathered from the IMDb movie review database. This data is split as shown in Figure 1 and used to train, develop, and test the models we use in this paper.

	Train	Dev	Test
Positive Reviews	12,500	6,250	6,250
Negative Reviews	12,500	6,250	6,250

Fig. 1. Dataset Split

The dataset is provided with 25,000 total training examples, and 25,000 testing examples. To perform our hyperparameter search, we further split the test set in half, using the first 12,500 examples of it for our dev set, and last 12,500 examples as our test set.

Before processing, punctuation marks such as commas, periods, and question/exclamation marks were filtered out using regular expressions [3]. However, stop words were left in place, and no stemming algorithm was used. The rationale for this was that our Word2Vec dictionary was large enough (400,000 words) to contain nearly all the various suffixes of words we encountered. Any word which could not be identified was mapped to the "unknown word" vector provided with the embeddings model.

The Naïve Bayes and SVM models used bag-of-words as features, with inputs encoded as a one-hot vector the size of the dictionary (400,000 words). The LSTM model used a pretrained Word2Vec word embeddings model from [3]. The word vectors were 50-dimensional.

For compatibility with the NumPy and TensorFlow input models (which require fixed-dimension input matrices), we truncate all movie reviews at 250 words. This truncation is typically not a problem, though, because the average movie review length (in the training set) is 233 words.

IV. MODELS

We constructed an LSTM RNN [3] to test our adversaries on. In addition to the LSTM, we built and trained a Support Vector Machine (SVM) and a Naïve Bayes model to benchmark LSTM in both clean test and adversarial conditions. The trained weights of the Naïve Bayes model were also used in the construction of two of the adversarial input generators we developed. The Naïve Bayes and SVM models were implemented using Python, Python3, and NumPy [8]. The LSTM model was implemented in TensorFlow [9]. All our results and findings are summarized in Section VI.

A. LSTM

Long short-term memories [1], or LSTMs, are a specific type of recurrent neural network well-suited for classification and prediction tasks on sequential data. LSTMs maintain a richer internal state representation than simple RNNs. For our project, we focused on a single-layer LSTM with a varying number of hidden units.

The LSTM layer has two inputs, which are the current element in the sequence, x_t, and its own internal state from the previous timestep, $state_{t-1}$. It then produces two outputs: the prediction out_t and its new internal state $state_t$.

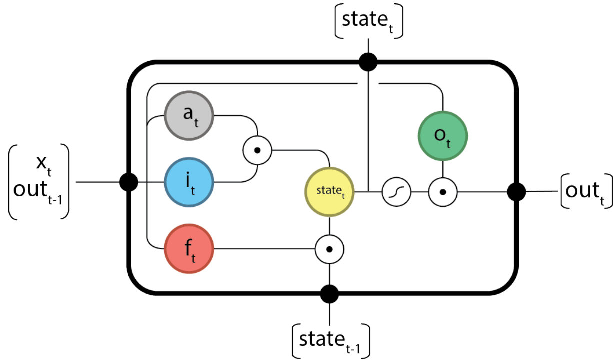

Fig. 2. An LSTM layer [10]. The symbol $\odot$ denotes element-wise multiplication.

The internals of the LSTM layer are defined by four components:

Input activation:
$$a_t = \tanh(W_a \cdot x_t + U_a \cdot out_{t-1} + b_a)$$
Input gate:
$$i_t = \sigma(W_i \cdot x_t + U_i \cdot out_{t-1} + b_i)$$
Forget gate:
$$f_t = \sigma(W_f \cdot x_t + U_f \cdot out_{t-1} + b_f)$$
Output gate:
$$o_t = \sigma(W_o \cdot x_t + U_o \cdot out_{t-1} + b_o)$$

which give us

Internal state:
$$state_t = a_t \odot i_t + f_t \odot state_{t-1}$$
Output:
$$out_t = \tanh(state_t) \odot o_t$$

The weights W and biases b for each gate are then trained via backpropagation [12]. Us are transition matrices, and define how hidden states should evolve across timesteps, similar to a Markov chain.

To classify an entire sentence, the LSTM layer is fed the words as input tokens across different timesteps. Then, the overall sentiment prediction is taken as the output of the LSTM upon seeing the final input token (word). Finally, a pooling layer provides additional weighting and biasing of the final output. This architecture of the LSTM cell "unrolled" across timesteps is given in Figure 3.

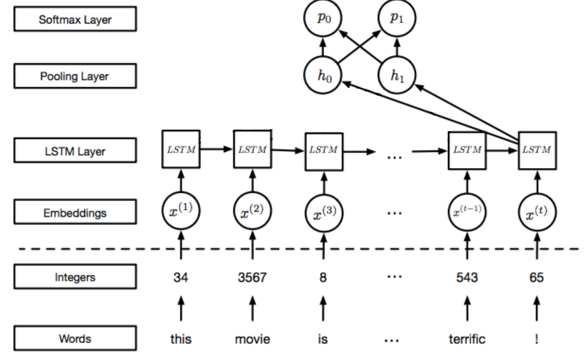

Fig. 3. The high-level LSTM architecture with Word2Vec.

The main focus of our work was to generate *adversaries* for RNNs; however, we did perform a hyperparameter search to optimize the LSTM being attacked. See Section VI for more information.

B. SVM

The SVM developed in this paper was based on [13] and implemented as Stochastic Gradient Descent on hinge-loss function [14] using scikit-learn library [17]. Input features used in SVM model were generated by a bag-of-words model and were in $|\mathcal{D}|$ dimensional space, where $\mathcal{D}$ is dictionary of words. A hyperparameter search was performed on learning rate and appropriate feature kernels (linear/polynomial) by first training the model on trainng set and then chosing the best model on dev set. The best model ended up being a linear SVM model trained with learning rate of 0.0001. It performed better compared to other two models in the non-adversarial setting.

We also tried some other input features for the SVM model to understand how input encoding affects SVM performance. We tried two new input encodings: mean Word2Vec for each review (50 dimensional vector) as well concatenating Word2Vec reviews back to back (50×250 dimensional vector). But both these feature sets performed significantly worse than bag-of-words model. Our understanding for this was that the first encoding reduced the information present in the review significantly as we were condensing the whole review to a single word. On the other hand, the second encoding resulted in a very high dimensional feature space over which SVM was no more efficient.

C. NAÏVE BAYES

The Naïve Bayes model used in this paper was developed from the derivation of multinomial Naïve Bayes with Laplace smoothing given in [11]. During training, every token (typically English words) of each of the training reviews is considered to be a training example $x^{(i)}$. We say there are m such tokens contained in the set of training reviews.

For each index j in the dictionary of words D, we compute the following during training:

$$\phi_{j|y=1} = \frac{1 + \sum_{i=1}^{m} 1\{x^{(i)} = \mathcal{D}[j] \wedge y^{(i)} = 1\}}{|\mathcal{D}| + \sum_{i=1}^{m} 1\{y^{(i)} = 1\}}$$

$$\phi_{j|y=0} = \frac{1 + \sum_{i=1}^{m} 1\{x^{(i)} = \mathcal{D}[j] \wedge y^{(i)} = 0\}}{|\mathcal{D}| + \sum_{i=1}^{m} 1\{y^{(i)} = 0\}}$$

Where $y^{(i)} \in \{0, 1\}$ is the label of the review that contains $x^{(i)}$, and $y = 0$ denotes a positive review, and $y = 1$ denotes a negative review.

The training procedure is carried out by observing all the training examples and maintaining two arrays (one array A_{neg} for negative reviews and one array A_{pos} for positive reviews) of size $|\mathcal{D}|$. These arrays store, at each index, the sum of 1 and the number of times each token is observed in positive (A_{pos}) or negative (A_{neg}) examples. After all training examples have been observed, $\phi_{y=0}$ is found by dividing each element of A_{pos} by the sum of $|\mathcal{D}|$ and the number of positive samples observed. Similarly, $\phi_{y=1}$ is found by dividing each element of A_{neg} by the sum of $|\mathcal{D}|$ and the number of negative samples observed. The value $\phi_{j|y=0}$ is then simply the jth index of $\phi_{y=0}$ (and likewise for $y = 1$).

Inference on a movie review R containing n words is conducted by first finding p_0 and p_1 where

$$p_0 = \prod_{i=1}^{n} \phi_{j|y=0}$$
$$p_1 = \prod_{i=1}^{n} \phi_{j|y=1}$$

and j is chosen for each term such that $R[i] = \mathcal{D}[j]$. The model then classifies R as being positive if $p_0 \geq p_1$ or negative if $p_0 < p_1$.

V. ATTACKS

We constructed three attacks as part of our work on this paper: Tack-On Adversarial Word, N Strongest Words Swap, and the Jacobian Saliency Map Approach (JSMA). The first two attacks are black-box approaches based on the weights learned by the Naïve Bayes model. The JSMA attack is a white-box approach developed in [15].

A. TACK-ON ADVERSARIAL WORD

Constructing this attack first required us to find the top five tokens with the strongest positive sway and the top five tokens with the strongest negative sway. We found these words by using the weights learned by our Naïve Bayes model; the set of tokens T_{pos} with the strongest positive sway is the set of tokens that maximize the ratio $\frac{\phi_{j|y=0}}{\phi_{j|y=1}}$, and the set of tokens T_{neg} with the strongest negative sway is the set of tokens that maximize the ratio $\frac{\phi_{j|y=1}}{\phi_{j|y=0}}$. Any ties that occurred were broken by choosing tokens in lexicographic order. The resulting five positive tokens were "edie," "antwone," "din," "gunga," and "yokai." and the resulting five negative tokens were "boll," "410," "uwe, "tashan," and "hobgoblins."

Once the most positive and most negative words were identified, we then constructed an adversarial dataset by modifying copies of the examples in our test set. The modifications we made were simple: replace the first token of all positive-labeled reviews with a token chosen randomly from the set of tokens with the strongest negative sway (and vice-versa for negative-labeled reviews).

B. N STRONGEST WORDS SWAP

The second adversary we developed was intended to be a direct improvement on Tack-On Adversial Word. As before, we insert adversarial tokens from T_{pos} and T_{neg} into reviews in an attempt to induce sentiment misclassification. Where Tack-On Adversarial Word replaced the first token in the review with an adversarial token, N Strongest Words Swap replaces N number of tokens in the review that have the strongest sway toward the correct classification. For this project, we constructed N Strongest Words Swap adversaries for N = 1, 3, and 5.

C. JACOBIAN SALIENCY MAP APPROACH (JSMA)

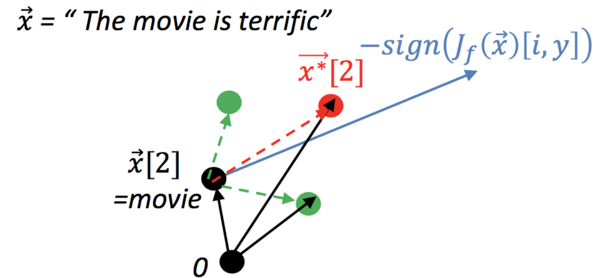

Fig. 4. Algorithm Illustration. For an example input sentence $\vec{x}$, suppose $\vec{x}[2]$ was selected. Then, blue arrow points at the direction of maximum increase in linearized approximation of loss function. With high probability none of the words will lie in this direction. As a heuristic, we choose the word which has highest projection along this direction and treat it as new adversarial word. This procedure is applied iteratively.

This section summarizes [15] and how we implemented the JSMA approach for generating adversaries in our LSTM example. In Jacobian Saliency Map approach, the idea is to explicit find out derivative of outputs wrt input and perturb the input in desired direction so as to selectively make the model mis-classify to an appropriate output class. More specifically, a Jacobian matrix J is constructed such that

$$J_f[i, j] = \frac{\partial f_j}{\partial x_i}$$

where x_i is the ith component of the input and f_j the jth component of the output. Then, perturbing x_i in $-sign(J_f[i,j])$ makes the cost function increase the most in order to decrease the outcome probability of f_j.

This approach is extended to RNNs (or LSTMs) by computational graph unfolding, i.e., instead of treating the LSTM as a sequential unit, we can construct a graph by explicitly keeping track of states at each time and then generate a Jacobian matrix like construction by taking derivatives of output wrt input at each time step. This is a tractable operation if number of unfoldings are finite as derivatives are straightforward to take in a recursive manner. This is like backprop step when training a Neural Net. More concretely,

$$J_f(\vec{x})[i,j] = \frac{\partial f_j}{\partial \vec{x}_i}$$

where Jacobian is now a vector wrt to each time sequence input to the network, and hence we know in which direction we need to perturb each word when $\vec{x}$ is a review sentence. Thus, we have overcome the *recursive* nature problem of RNNs.

For overcoming the *discrete* nature of $\vec{x}$, a heuristic is used to find the word vector whose projection along this perturbation is **maximum**. The whole process of selecting a word and finding an adversary is done iteratively until an adversarial example is found or some upper limit of M words are perturbed. A large value of M results in larger running time and a smaller value of M results in fewer success in adversarial example generation. Figure 4 gives the overview of the algorithm and Figure 5 shows the pseudocode for implemented algorithm.

Input: *f*, $\vec{x}$, *D*
Algorithm:
1. $y := f(\vec{x})$
2. $\vec{x^*} := \vec{x}$
3. $J_f(\vec{x})[y] = \frac{\partial h_y}{\partial \vec{x}}$
4. *while* $f(\vec{x^*}) == y$:
5. *select a word* ***i*** *in sequence* $\vec{x^*}$
6. $\vec{w} := armin_{\vec{z}\epsilon D} \left|sign(\vec{x^*} - \vec{z}) - sign(J_f(\vec{x})[i,y])\right|$
7. $\vec{x^*}[i] := \vec{w}$
8. *end*
9. *return* $\vec{x^*}$

Fig. 5. The JSMA algorithm. Inputs are f: Prediction Model, $\vec{x}$: Input Sentence, D: Dictionary. Output is $\vec{x^*}$: Adversarial Sentence

Specifically, in our implementation we used TensorFlow to generate gradient graph and we took gradient at the pooling layer instead of softmax layer. This helps to overcome gradient saturation that could happen at the softmax layer. Since LSTMs have finite memory, most-recent words before prediction were chosen to be perturbed. To compute step 6 in the algorithm, an efficient vectorization and matrix product were used to find the word in the dictionary with maximum projection on perturbation direction. Due to lack of computation power we restricted our experiment to changing a maximum of $M = 35$ words out of 250 word review limit only running this attack on a subset of the dataset.

VI. RESULTS & DISCUSSION

A. *MODELS*

Before attacking the LSTM with the three adversaries, we performed a basic hyperparameter search to optimize clean accuracy. One key hyperparameter for LSTMs is the number of hidden units, which corresponds to the size of the weight matrices for the gates, and thus the amount of internal state maintained. We compared 32-, 64-, and 128-hidden-unit LSTMs, and found that the 64-unit LSTM performed best on the dev. set (see Figure 6). Thus, we chose it for our further experiments.

Our hypothesis is that the 32-unit LSTM wasn't expressive enough to model the complexity of the reviews, and the 128-unit LSTM overfit the training set. This hypothesis is supported by the fact that while training set accuracy for the 32-unit LSTM took the full 100K iterations to converge, the 64-unit LSTM took 60-70K, and 128-unit took 30-40K. Each model took around 8 hours to train, with higher-unit models taking slightly longer.

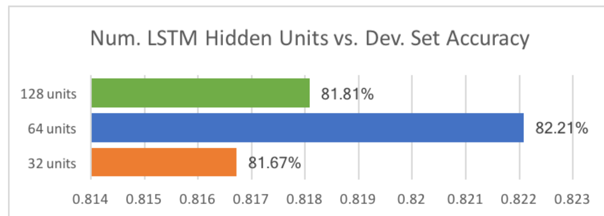

Fig. 6. Sweeping num. LSTM hidden units.

To further address overfitting, our LSTM training architecture included dropout on the hidden units. A dropout rate of 50% proved inferior to a rate of 25%, with 81.77% and 82.21% dev. set accuracies, respectively (for 64 hidden units). The Adam optimizer [5] and a training batch size of 24 examples were used for all models.

To further explore the training set, we performed principal component analysis on it. As seen below in Figure 7, even just two principal components demonstrate class separation [17]. This may suggest why our linear SVM and Naïve Bayes models were able to perform as well as they did.

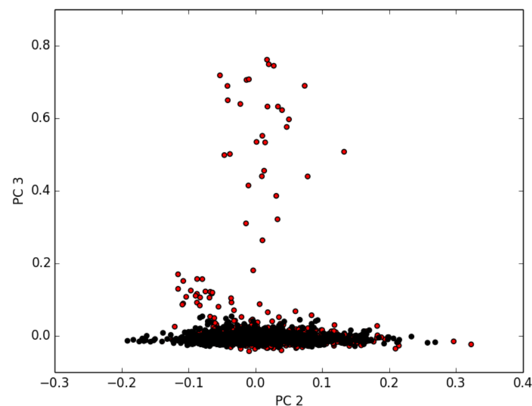

Fig. 7. Principal Component Analysis of the IMDb dataset.

B. ATTACKS

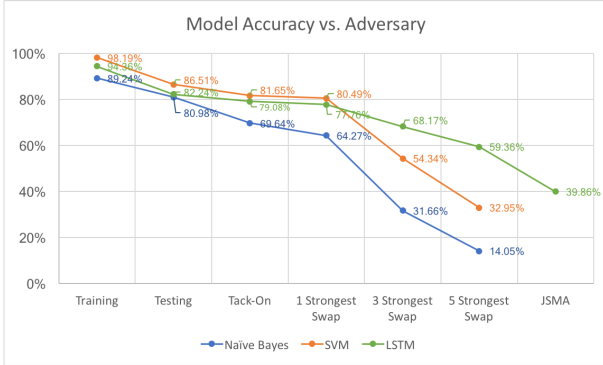

Fig. 8. Accuracy of models when subject to each set of inputs.

When choosing which adversaries to construct, we considered two approaches : black-box and white-box. We anticipated that a black-box approach, requiring no knowledge of model parameters (Tack-on adversary and N strongest word swap), would be quicker and simpler to build but have reduced impact on the model's performance. In contrast, white-box approaches (JSMA) should take more time to construct but have higher impact, given their specialized nature. What we hypothesized was correct, but the extent to which the black-box adversaries affected the LSTM exceeded our expectations.

All three adversaries had negative impact on LSTM performance. Tack-On Adversarial Word and 1 Strongest Word Swap had almost negligible impact on LSTM performance, but 3 Strongest Words Swap, 5 Strongest Words Swap, and JSMA were able to bring test accuracy down from 82.24% to 68.17%, 59.36%, and 39.86%, respectively. As seen in Figure 8, LSTMs are more robust to adversaries than other models and accuracy of all models fall monotonically with increasing adversary strength.

Figure 9 shows that JSMA was able to target and sway both positive and negative sentiments proportionally. Even more interesting is Figure 10 which shows that in many successful adversarial examples we only need to change a few words!

Another point that we found interesting was that the adversarial words utilized by all three models seemed to have little to no effect on our interpretation of positive and negative sentiment. We were anticipating words like "excellent" and "horrible" to be the tokens that were most indicative of sentiment. This was not the case. We saw our adversarial input generators using tokens like "brute," "sonogram," etc. to perturb reviews toward misclassification. This leads us to question what exactly the models are learning and using to classify sentiment.

	y=Positive	y=Negative
True Positive	54	65
True Negative	110	62

Fig. 9. Confusion Matrix after JSMA attack on LSTM: JSMA is unselective towards positive and negative sentiments and effects both of them similarly.

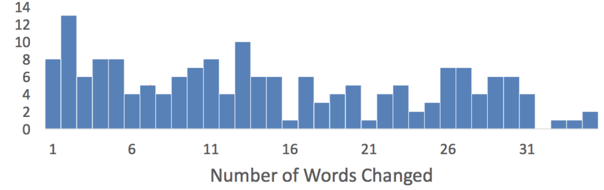

Fig. 10. Number of words replaced by JSMA method in the cases where adversarial examples were successfully created. A lot of examples exist where only changing a few words resulted in adversary generation.

VII. FUTURE WORK

One limitation of our work is the adversarial sentences generated are often unintelligible to a human. For example, the JSMA adversary perturbed the sentence ***This excellent movie made me cry!*** to ***This excellent tsunga telsim grrr cry***. The intent of our work was to show that adversarial examples can be generated in a mechanical way, and indeed, even nonsensical examples might be useful in fooling, for example, spam detection models for forum posts. However, an interesting direction for future work would be to introduce a proper statistical NLP parser such as [6] into the loop. After the adversary perturbs its example, the parser could check it for grammatical correctness. A variety of schemes could be built on top of the feedback provided by the parser; most obvious amongst these is a reinforcement learning setup which uses parser hints to inform the adversarial generation. GANs [7] are another interesting candidate for adversaries.

Beyond this, LSTM with mean-pool layer or a multilayer LSTM architecture might provide better test set accuracy, and perhaps better resilience to adversaries. We would like to train these and check this hypothesis. We would also like to do optimized memory allocation in TensorFlow code for JSMA method and run it on GPUs. Finally, throughout our work, all adversarial examples were evaluated only in the test (inference) phase. In the future, we would like to experiment to see how adversarial training could be used to develop more robust models by analyzing the weaknesses of the models.

VIII. CONCLUSION

In summary, we developed an LSTM binary sentiment classifier, as well as Naïve Bayes and SVM baselines. We then evaluated these models' susceptibilities to three distinct types of adversarial attacks: Tack-On Adversarial Word, N Strongest Words Swap, and JSMA. We found that although our single-hidden-layer LSTM loses slightly in test set accuracy to the SVM, it is much more resilient to the adversarial attacks, and outperforms both baselines in the presence of any given adversary.

As we anticipated, the white-box JSMA method produced the most successful adversarial examples. However, even the simple N Strongest Word Swap method outperformed our expectations by fooling LSTMs easily.

IX. ACKNOWLEDGEMENTS

We would like to acknowledge many helpful discussions with Aditi Raghunathan, Rahul Trivedi and Jeremy Irvin.

X. CONTRIBUTIONS

We each led development of one machine learning model on the sentiment analysis dataset: Andy developed the LSTM model, Pulkit developed the SVM, and Mark developed the Naïve Bayes model. All members worked together to develop the tack-on-word adversary. Andy refined the LSTM with hyperparameter search, Mark developed the N-strongest-words-swap adversary, and Pulkit developed the JSMA adversary. All members contributed equally to writing the poster and reports.

REFERENCES

[1] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):17351780, November 1997.

[2] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142150, Portland, Oregon, USA, June 2011. Association for Computational Linguistics.

[3] Adit Deshpande. Sentiment Analysis with LSTMs. https://github.com/adeshpande3/ LSTM-Sentiment-Analysis.git, 2017.

[4] Nicolas Papernot, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Fartash Faghri, Alexander Matyasko, Karen Hambardzumyan, Yi-Lin Juang, Alexey Kurakin, Ryan Sheatsley, Abhibhav Garg, and Yen-Chen Lin. cleverhans v2.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 2017.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014

[6] The Stanford Parser: A Statistical Parser. https://nlp.stanford.edu/software/lex-parser.shtml

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, Generative adversarial nets, *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), Curran Associates, Inc., 2014, pp. 26722680.

[8] NumPy. http://www.numpy.org/.

[9] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Vigas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] Aidan Gomez. Backpropagating an LSTM: A Numerical Example. https://blog.aidangomez.ca/2016/04/17/Backpropogating-an-LSTM-A-Numerical-Example/

[11] Andrew Ng. CS 229 Lecture Notes Part IV: Generative Learning Algorithms. http://cs229.stanford.edu/notes/cs229-notes2.pdf

[12] Andrew Ng. CS 229 Lecture Notes Deep Learning: Backpropagation. http://cs229.stanford.edu/notes/cs229-notes-backprop.pdf

[13] Andrew Ng. CS 229 Lecture Notes: SVM. http://cs229.stanford.edu/notes/cs229-notes3.pdf

[14] John Duchi CS 229 Supplementary Lecture Notes: Loss Functions. http://cs229.stanford.edu/extra-notes/loss-functions.pdf

[15] Nicolas Papernot, Patrick D. McDaniel, Ananthram Swami, and Richard E. Harang, Crafting adversarial input sequences for recurrent neural networks, CoRR abs/1604.08275 (2016).

[16] Weiwei Hu and Ying Tan. Black-Box Attacks against RNN based Malware Detection Algorithms.arXiv (2017)

[17] scikit-learn. http://scikit-learn.org/stable/