Project Machine Learning
— Milestone 2 —

Johannes Constantin Keller, Moritz Wassmer, Ivan Akunevich

September 3, 2024

1 Introduction

Last Milestone, we focused on implementing the vanilla BERT architecture to solve the toxicity classification task. Besides the implementation of the architecture itself, we also had to adapt the algorithms to account for multi-label classification and class imbalance. This involved choosing a feature extraction technique to convert the text into a numerical representation and defining a loss function. We also chose appropriate evaluation metrics to perform model selection. The implemented model was able to converge and showed promising results.

This Milestone is going to focus on model selection and evaluation. First, we discuss all relevant hyperparameter choices. Afterwards, we will present results of the model selection optimization and estimate the generalization performance. Finally, we are going to discuss the results.

2 Methodology

In Milestone 1, we considered several methods for solving the task. For instance, we considered various ways to convert text into vectors and several techniques to solve the multi-label classification. In this chapter, we will describe and discuss our hyperparameter choices.

2.1 Fixed Hyperparameters

The BERT neural network architecture also provides tunable hyperparameters such as model size and dropout. Since exploring all possible hyperparameter combinations breaks the scope of this project, we fixed some of these choices. In this section, we explain which hyperparameters we keep fixed in all experiments.

Model Size When using BERT, various configurations for model size are possible. The main architectural choices involve the dimensionality of the word vectors H, the amount of encoder layers L, the amount of attention heads A, and the maximal sequence length S. In general, increasing the model size leads to continual improvements, even on small scale tasks (Devlin et al., 2019). To reduce the computational complexity, we chose the *BERT-Base* configuration with the dimensions ($H = 768, L = 12, A = 12, S = 512$). We skipped the pre-training procedure by loading pre-trained weights from huggingface, which supports a maximum of $S = 512$ tokens.

The effect of the different parameters can be described as follows: Adding more encoder layers L to BERT generally enhances the model's capacity to capture intricate hierarchical features and contextual information within the input text. Adding attention heads A helps at detecting more diverse patterns in input sequences, potentially improving contextual representation. In contrast to only using one set of weights for extracting features, *multi-head attention* employs multiple sets of weights, called *attention heads* in parallel. All attention heads extract relationships independently. However, adding encoder layers and adding attention heads comes both at the cost of increased computational complexity (Vaswani et al., 2017).

Pre-Processing and Embedding Method We implemented the original pre-processing and embedding, called *WordPiece-Embeddings* (Wu et al., 2016). That means, that a word is split into pairs of characters. This allows the model to learn the meaning of word compounds instead of entire words. These tokens are then transformed into a H-dimensional vector in the embedding layer. After adding a positional and segment embedding, the resulting vectors are then processed by the BERT encoder layers. The vocabulary and configuration of the WordPiece tokenizer are also loaded from the pre-trained version. The tokenizer converts all characters strings into their lowercase variants before converting them into tokens. The vocabulary size is 30522 tokens. The tokenizer ensures the maximal sequence length of the tokens S is not exceeded. In Milestone 1, we examined the distribution of token lengths in the training dataset. Our analysis revealed that the sequence length distribution is centered significantly below 512, with a 75-percent-quantile ($Q_{0.75}$) at approximately 100. Therefore, we expect the sequence length to be sufficient for the problem. Nevertheless, if the input sequence exceeds the sequence length, tokens beyond that specific length are discarded. If the number of tokens is below the maximal sequence length, padding tokens fill the remaining positions to indicate no available tokens. The first token is always the classification token (CLS), which is added at the very front of the tokens. It is used later to obtain a representation which aggregates information of the whole input sequence.

Dropout Dropout layers help at reducing overfitting. The dropout rate p determines the rate of neurons which are randomly selected during training to be dropped. Dropping a neuron means temporarily deleting all incoming and outgoing connections (Srivastava et al., 2014). A higher dropout rate results in higher regularization. We chose to use $p = 0.1$, consistent with both the original paper Devlin et al. (2019) and the paper, which focuses on fine-tuning BERT (Sun et al., 2019).

Optimizer Our chosen optimizer is Adam, as introduced by Kingma and Ba (2015), which is used in the original implementation as well as in Devlin et al. (2019) and Sun et al. (2019). Adam updates the gradient based on information of past gradients by calculating moving averages (first order raw moments) and squared moving averages (second order raw moments) of the gradients. This configuration is also suggested by Kingma and Ba (2015) as being effective for the tested tasks.

The original Adam algorithm has three key hyperparameters: the learning rate, which is described in section subsection 2.2, β_1 and β_2. We utilize $\beta_1 = 0.9$ and $\beta_2 = 0.999$, which is recommended in all three mentioned papers. Both parameters determine the weight assigned to past gradient information when updating the weights during learning. Smaller values of β_1 and β_2 make the algorithm more responsive to recent updates, while larger values provide more smoothing and stability over time.

Multi-Label Classification One key decision is which method to use to account for multiple labels being present simultaneously. In Milestone 1, we considered different methods of incorporating this property into our learning algorithms in more detail. The *transformation into a multi-task learning problem* method seemed the most promising in terms of complexity and performance. It has been shown that learning multiple tasks in parallel utilizing shared representations can improve generalization by using the domain information contained in the training signals of related tasks as an inductive bias (cf. Caruana (1997)). We implemented this by adjusting the neurons to the number of classes, similar to multi-class classification. But instead of using a single *cross-entropy loss* we used individual *binary-cross-entropy loss* functions for each binary class which we combine by averaging the losses. The formula is detailed in section 3.

Compared to the other methods, it is computationally efficient since there is no need to train multiple classifiers as required by classifier chains or the binary relevance method. At the same time, we avoid having a large number of permutations of labels when transforming the task into a multi-class classification problem. This way we simplify interpretation and implementation of the solution.

Class Imbalance To reduce class imbalance, *sampling strategies* such as *under-* or *oversampling* can be considered. The advantage of undersampling is that computational complexity is reduced by decreasing the size of the dataset. At the same time, it reduces

the amount of information the model can learn (Johnson and Khoshgoftaar, 2019). Oversampling makes use of the full dataset by replicating underrepresented classes. However, given the large class imbalance, the resulting dataset would grow substantially (Johnson and Khoshgoftaar, 2019). We decided to stick to assigning weights to the positive classes by adjusting the loss function. This has the advantage of utilizing the full diversity of the dataset while being computationally feasible. The formula for calculating the class weights and their incorporation into the loss function can be found in section section 3.

Batch Size The batch size refers to the number of training examples utilized for one update of the weights. It is a hyperparameter that impacts training performance. Larger batch sizes may lead to faster training but require more memory, while smaller batch sizes often prolong training but can enhance generalization (Keskar et al., 2016).

We use a batch size of 16, a value recommended in the original BERT paper that proved effective across all tested tasks (Devlin et al., 2019).

2.2 Hyperparameter Optimization

In this section, we introduce all methods that we consider for hyperparameter optimization. We follow both the suggestions for fine-tuning from the original BERT-paper (Devlin et al., 2019), as well as suggestions from Sun et al. (2019).

To test different methods, we split the data into train, test and validation sets in the following proportions: 70 percent for training, 15 percent each for testing and validation. All methods use the first two sets for training and evaluation, the best performing combination of hyperparameters is validated with the last set.

2.2.1 Methods

We consider two different ways of scheduling the learning rates. The first one, called *baseline* or *slanted triangular learning rates*, follows a similar approach as presented in the original implementation of the BERT paper (Devlin et al., 2019). The second approach introduces modifications as suggested by Sun et al. (2019).

Baseline As proposed by Howard and Ruder (2018), model parameters can be adapted to task-specific features. This can be achieved by converging the model to the corresponding region of the parameter space directly in the beginning of training, afterwards parameters are refined. As stated in the paper, the usage of the same learning rate (LR) or an annealed learning rate throughout training might not be a suitable solution to achieve the required behaviour. Hence, *slanted triangular learning rates* (STLR) are proposed: The algorithm linearly increases the learning rate first and then linearly decays it according to the schedule, which is defined as:

$$\begin{aligned} cut &= \lfloor T \cdot cut\_frac \rfloor \\ p &= \begin{cases} t/cut, & \text{if } t < cut \\ 1 - \frac{t - cut}{cut \cdot (1/cut\_\text{frac} - 1)}, & \text{otherwise} \end{cases} \\ \eta_t &= \eta_{max} \cdot \frac{1 + p \cdot (ratio - 1)}{ratio} \end{aligned} \tag{1}$$

where T is the number of training iterations, *cut_frac* is the fraction of iterations during which the learning rate is increased, *cut* is the iteration when the learning rate is switched from increasing to decreasing, and p represents the proportion of iterations fro which the learning rate has been increased or will be decreased. The *ratio* shows how much smaller the lowest learning rate is compared to the maximum learning rate $\eta_{\max}$, and $\eta_t$is the learning rate at iteration t.

It is important to mention, that the *slanted triangular learning rate* is similar to the implementation of the original paper Devlin et al. (2019). The authors describe the process as warm-up learning rate followed by a linear decay, which is essentially what SLTR is all about. SLTR is called the *baseline* in this Milestone.

Slanted Triangular Discriminative Fine-Tuning Since the different layers of a neural network can capture different types of information, for example deeper layers usually contain general concepts, they should be fine-tuned differently. That is why the *discriminative fine-tuning* method (Howard and Ruder (2018)), which extends the capability of the *baseline* approach, can be used to address this issue.

To tune layers with different learning rates, the parameters θ are grouped into $\{\theta_1, \ldots, \theta_L\}$ where θ_l contains the parameters of the l-th layer of BERT. As an example, to update parameters using the stochastic gradient descent, the following formula is used:

$$\theta_t^l = \theta_{t-1}^l - \eta^l \cdot \nabla_{\theta^l} J(\theta), \tag{2}$$

where η^l represents the learning rate of the l-th layer.

The base learning rate is η^L and we use $\eta^{k-1} = \xi \cdot \eta^k$, where ξ is a decay factor. When $\xi < 1$, the deeper layer has a lower learning rate than the upper layer. If we set ξ to 1, the solution is equal to regular stochastic gradient descent (SGD), meaning that all layers have the same learning rate.

2.2.2 Learning Rates

Intuitively, it would be reasonable to start with learning rates as proposed in the papers mentioned above. But unfortunately, the loss functions are not comparable to the ones proposed. In both papers, for fine-tuning they used balanced datasets. Therefore, they do not assign weights to the classes in the loss function. We are using an unbalanced dataset and decided to address the class imbalance problem by using a weighted loss function. We assign a higher weight > 1 to the positive classes while refraining from manipulating negative classes. While the proportions for the expected loss by positive and negative classes are balanced this way, our loss tends to be higher in general. Therefore we do need to use a smaller learning rate than proposed in the literature.

Additionally, the BERT model faces *catastrophic forgetting* (McCloskey and Cohen (1989)), which is a common problem in transfer learning. This means that during learning new knowledge (fine-tuning for specific task), the pre-trained knowledge can be lost. To solve this issue, a lower learning rate is commonly used. As mentioned by Sun et al. (2019), high learning rates lead to convergence failures. During the training phase we also observed, that lower learning rates helped to achieve more stable and accurate results.

Combining these two factors, we chose learning rates in the range from 3e-5 to 1e-6, meaning that the learning rate range is smaller than proposed in original paper and close to the suggested ones in Sun et al. (2019).

2.2.3 Epochs

The number of epochs indicates, how many times the learning algorithm will go through the entire training dataset. It is important for model convergence, meaning that model's performance cannot be significantly improved. Wise choice of this parameter also allows to control underfitting or overfitting of the model. Following the original implementation, Devlin et al. (2019) and Sun et al. (2019), we used the range from one to four epochs as a hyperparameter.

3 Empirical Estimate of the Generalization Error

This section introduces important applied error and performance measures and evaluates and compares the methods' performances based on these.

3.1 Error Measure: Binary Cross-Entropy Loss

We can not express the loss using a normal cross-entropy loss since the labels are not exclusive. Therefore, we compute the BCE loss for each output neuron individually.

In Equation 3, $\ell_{n,c}$ represents the BCE per class c, where n is the number of labels per batch. A *sigmoid function* (σ) is applied to each component to obtain a probability for this class.

$$\ell_{n,c} = -\left[p_c y_{n,c} \cdot \log \sigma(x_{n,c}) + (1 - y_{n,c}) \cdot \log(1 - \sigma(x_{n,c}))\right] \tag{3}$$

To counteract the class imbalance, we incorporated weights (p_c) into the loss function for the positive classes. Consequently, we penalize errors on rare classes:

$$p_c = \frac{dataset\ size}{(number\ of\ comments\ labeled\ c) \cdot (number\ of\ classes)} \tag{4}$$

We modified the manner of calculating the class weights compared to Milestone 1. We multiplied the weights from Milestone 1 with $\frac{1}{number\ of\ classes}$ to account for having multiple classes. We hope that we have more losses with magnitudes comparable to the literature, in which usually balanced datasets instead of class weights are used.

3.2 TPR and FPR Tradeoff: ROC-AUC

The decision regarding the right balance of FPR or FNR is task specific and individual. A higher FPR might be preferable to a higher FNR, since the latter implies missing toxic comments, which could cause harm to individuals. On the other hand, deleting a comment without a good reason might hurt the freedom of speech. The tradeoff can be tuned by adjusting the classification threshold.

We decided to evaluate our methods with a more general performance measure, called AUC-ROC (*Area Under The Curve - Receiver Operating Characteristics*), as described in **?**. The concept of AUC-ROC is to evaluate TPR and FPR across various thresholds and aggregate the performances in a single score. The aggregation is performed by approximation of the area under the ROC-curve, hence the name. The AUC value ranges between 0.0 (indicating, the model predicts the opposite class of the label), over 0.5 (it cannot separate between the two classes) to 1.0 (it can separate them perfectly). The formula of TPR and FPR are shown in Equation 5.

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \qquad \text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \tag{5}$$

We calculate the AUC-ROC score for each class individually and compute an average to receive a total score for model selection.

3.3 Results of the Methods

In this section, we will discuss the outcomes of the two methods we applied and compare and interpret them regarding the chosen error measure.

BCE-Loss Figure 1 and Figure 2 display the BCE-error on the test dataset using models with different learning rate configurations. They show the loss for the best- and the worst-case learning rate, as well as the median across all testing epochs for each of the scheduling methods, respectively.

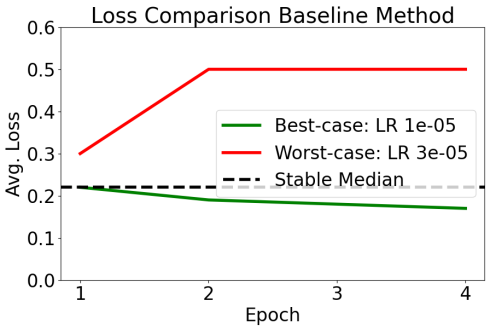

Figure 1: Avg. test epoch BCE loss of baseline for best-case and worst-case learning rate.

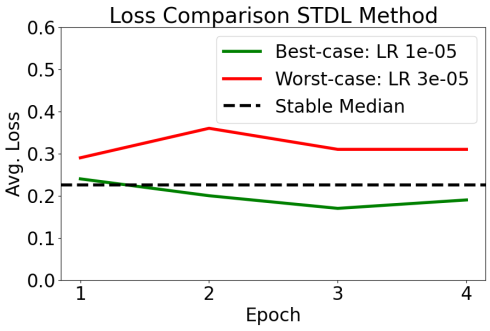

Figure 2: Avg. test epoch BCE loss of STDL for best-case and worst case learning rate.

In general, the error of STDL was less sensitive to the learning rate compared to the baseline's error. However, both STDL and the baseline achieved the optimal value among all runs, which is 0.17, at the same learning rate of $1e-05$. STDL and the

baseline performed similarly well, with both exhibiting a median generalization error of 0.23 (see Figure 2) and 0.22 (see Figure 1)respectively.

ROC-AUC Figure 3 and Figure 4 illustrate the ROC-AUC values for the best- and the worst-case learning rates of the respective scheduling methods.

Regarding the ROC-AUC, the STDL method also showed greater stability. Nevertheless, both scheduling methods achieved the optimal value of 0.98 at a learning rate of $1e-05$. The median was 0.97; indicating a consistently high performance. The ROC-AUC dropped to its lowest value of 0.50 when using baseline scheduling with a learning rate of $3e-05$ (see Figure 3). We attribute this to potential catastrophic forgetting, indicating that the learning rate is set to high. This assumption is supported by the fact that smaller learning rates achieved substantially better performances. STDL exhibited more stability at the same learning rate ($3e-05$) because the effective learning steps for all weights across all epochs are way smaller. The learning rates are lower for two reasons: First, the discriminative learning rate scheduling means deeper layers operate on lower learning rates. Second, the learning rate reaches only $3e-05$ at the peak; it is lower before (warm-up phase) and afterwards (decay phase).

Overall, the results based on the ROC-AUC performance measure align with the results of the loss measure in section 3.3. This indicates that the loss function has been chosen correctly and aligns well with the goal of maximizing the ROC-AUC.

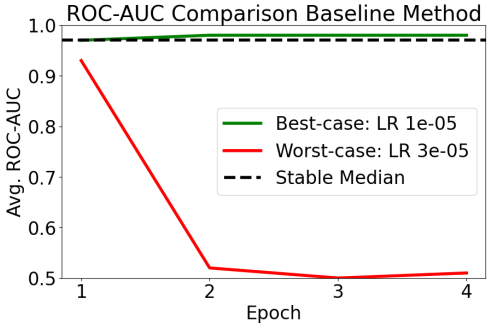

Figure 3: Avg. testing epoch ROC-AUC of the baseline for the best-case and worst-case learning rates.

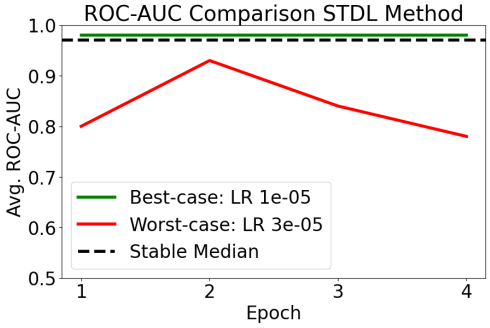

Figure 4: Avg. testing epoch ROC-AUC of STDL for the best-case and worst-case learning rate.

4 Discussion

Training Expenses Considering time, the expenses associated with training the model are relatively low. Since we skip the pre-training procedure by loading pre-trained weights we only need to fine-tune the model. When using a low learning rate, both methods converge within only a handful of epochs. The slanted learning rate scheduling yields fast convergence due to its exploratory phase in the beginning and the exploitative phase in the end of the training (cf. Howard and Ruder (2018)). On a NVIDIA A100 80GB PCIe, this translates for a big data set of 150000 examples to a training time of little more than two hours.

Considering hardware, the expenses associated with training are rather high. To train the algorithm in an efficient manner, a high memory GPU is necessary. Hardware requirements can be lowered though by choosing a smaller batch-, or model-size. Nevertheless, the performance is likely going to differ when adjusting these parameters.

Confidence Measure A confidence measure per class can be determined by employing the *Shannon-based confidence* (cf. Bukowski et al. (2021)). The Shannon-based confidence of a n-dimensional probability vector v_n is defined as:

$$Conf_{Shannon}(v_n) = 1.0 - H_n(v_n) \tag{6}$$

The entropy $H_n(v_n)$, in our case represented as H_2, is defined for the two-dimensional per-class probability vector $v_2 = (p, (1-p))^T$ as:

$$H_2 = -[p \cdot log(p) - (1-p) \cdot log(1-p)] \tag{7}$$

For a given label, p represents the model's output probability for the label's presence, and $(1-p)$ the probability for this label's absence per instance. The closer the Shannon-based confidence is to one, the more confident is the model about its classification. Whereas the closer to zero, the less confident the model becomes. This computation is only conducted for instances where the label is *True* and averaged across the total count of *True* labels per class.

Table 1 shows the averaged Shannon-confidences associated with the respective classes, obtained during the validation phase of the hyperparameter optimization for the baseline and the STDL method.

Confidence for:	Toxic	Severe Toxic	Obscene	Threat	Insult	Identity hate
Baseline	0.91	0.91	0.91	0.95	0.86	0.93
STDL	0.92	0.93	0.93	0.98	0.89	0.95

Table 1: Averaged Shannon-based confidences for classes of toxicity for the methods baseline and STDL.

The highest average confidence of 0.98 was achieved by STDL for the class "threat", meaning the model has a confidence of 98 percent on average that it can classify an instance of this class correctly. The high confidences of both methods for this class may be due to its recognizability by specific keywords. All the other classes do not fall below 86 percent for both methods, ranging on average over 90 percent (see Table 1). Additionally, STDL's confidences are consistently higher compared to the baseline's.

Conclusion We compared the performance of the baseline method, which utilizes a slanted triangulated learning rate scheduler, to the STDL method, which uses additive discriminative learning rate scheduling, for toxic comment classification. The main difference to comparable settings in the literature, such as Sun et al. (2019), is the imbalance of our dataset, the class weights we apply due to this imbalance, and the multi-label classification setting.

Comparing STDL and the baseline, both are equally fast at achieving convergence. With an appropriate learning rate, both methods present a similar ROC-AUC. However, STDL is less sensitive to the chosen learning rate. The reason for the stability of the discriminative approach is that it is more capable of fine-tuning its predictions without forgetting the pre-training. Comparable to Sun et al. (2019), the upper layers might be primarily responsible for the classification and are affected more by the gradient, whereas the deeper layers, primarily responsible for general understanding, are not influenced as much. Given that layer-wise discriminative learning rate scheduling is proposed as a countermeasure to catastrophic forgetting by Sun et al. (2019), it is not surprising that we only saw catastrophic forgetting only in the baseline.

Comparing our result of ROC-AUC of 0.98, we achieve similar performance to the top of the Kaggle leaderboard[1] , which achieves a performance of 0.98856. Considering that we only used 70% of the training data that they used, since they evaluated on a hidden test set, this can be considered a great performance.

All in all, we found BERT with our configuration to be an excellent choice for solving the task of classifying toxicity. It achieved low error rates, while being fast to train. The only sensitive parameter seems to be the learning rate. We recommend to set the learning rate smaller than the recommended values in the literature when using weights for positive classes.

[1] https://www.kaggle.com/competitions/jigsaw-toxic-comment-classification-challenge/leaderboard

References

M. Bukowski, J. Kurek, I. Antoniuk, and A. Jegorowa. Decision confidence assessment in multi-class classification. *Sensors*, 21(11), 2021. ISSN 1424-8220. doi: 10.3390/s21113834. URL https://www.mdpi.com/1424-8220/21/11/3834.

R. Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, jul 1997. ISSN 0885-6125. doi: 10.1023/A:1007379606734. URL https://doi.org/10.1023/A:1007379606734.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

J. Howard and S. Ruder. Universal language model fine-tuning for text classification. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 328–339, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-1031. URL https://aclanthology.org/P18-1031.

J. M. Johnson and T. M. Khoshgoftaar. Survey on deep learning with class imbalance. *Journal of Big Data*, 6:1–54, 2019. URL https://api.semanticscholar.org/CorpusID:102354936.

N. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 09 2016.

D. Kingma and J. Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, San Diega, CA, USA, 2015.

M. McCloskey and N. J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier, 1989.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

C. Sun, X. Qiu, Y. Xu, and X. Huang. How to fine-tune bert for text classification? In M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu, editors, *Chinese Computational Linguistics*, pages 194–206, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32381-3.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.