Transformer with Memory Replay

Rui Liu **Barzan Mozafari**

Computer Science and Engineering
University of Michigan, Ann Arbor
{ruixliu, mozafari}@umich.edu

Abstract

Transformers achieve state-of-the-art performance for natural language processing tasks by pre-training on large-scale text corpora. They are extremely compute-intensive and have very high sample complexity. Memory replay is a mechanism that remembers and reuses past examples by saving to and replaying from a memory buffer. It has been successfully used in reinforcement learning and GANs due to better sample efficiency. In this paper, we propose *Transformer with Memory Replay* (TMR), which integrates memory replay with transformer, making transformer more sample-efficient. Experiments on GLUE and SQuAD benchmark datasets show that Transformer with Memory Replay achieves at least 1% point increase compared to the baseline transformer model when pretrained with the same number of examples. Further, by adopting a careful design that reduces the wall-clock time overhead of memory replay, we also empirically achieve a better runtime efficiency.

Introduction

Transformers have achieved state-of-the-art performance on various natural language processing (NLP) tasks, such as sentimental anaylysis, paraphrase detection, machine reading comprehension, text summarization, question answering and so on (Devlin et al. 2018; Liu et al. 2019b; Dai et al. 2019; Brown et al. 2020). The training of tranformers typically consists of two stages: pre-training and fine-tuning. Pre-training is the stage of training a generic model on an enormous corpus, such as Wikipedia to learn the representation inherent in understanding the natural language. Fine-tuning is the stage of training a task-specific model that is initialized with pre-trained parameters on the dataset for the specific downstream task for just a few epochs. Each downstream task has a separate fine-tuned model, even though it is initialized with the same pre-trained parameters. To get good generic representation, transformers are usually very large models with a huge number of parameters. For example, OpenAI recently released GPT-3, which contains 175 billion parameters (Brown et al. 2020). Training these large-scale models is extremely compute-intensive and has very high sample complexity for the pre-training stage (Devlin et al. 2018; Clark et al. 2020). To make transformers more sample efficient, Clark et al. (2020) proposed a new transformer model, called ELECTRA, that consists of two modules: generator and discriminator. Both generator and discriminator are transformers, although generator is usually smaller in size. The discriminator is trained to predict whether each token in the corrupted input was replaced by the generator. They showed that their model substantially outperforms previous models, such as BERT and XLNet given the same amount of data.

In this paper, we go one step further by integrating the memory replay mechanism into ELECTRA, which we show can further improve the sample efficiency. Memory replay works by maintaining a fixed-size memory buffer that holds the most recent examples. It greatly improves the sample efficiency by enabling examples to be reused multiple times for training, rather than throwing away examples immediately after one-time usage. By controlling the strategy on how examples are managed in the memory buffer (e.g., how to assign weights to examples), memory replay can be customized according to specific needs. Memory replay mechanism has been successfully used in reinforcement learning [1] and GANs, because of its improvement on sample efficiency (i.e., requires less amount of samples to achieve the same accuracy) (Schaul et al. 2015; Fedus et al. 2020; Wang et al. 2016; Wu et al. 2018).

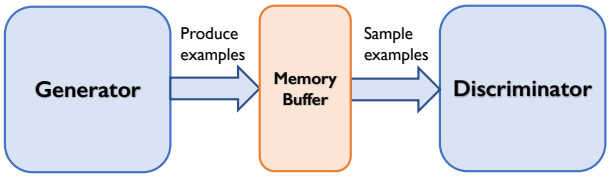


Figure 1: The architecture diagram of our *Transformer with Memory Replay* (TMR). The generator produces corrupted examples, which are being saved to the fixed-size memory buffer. The discriminator samples examples from the memory buffer for training. High-quality examples are reused by the discriminator, thus making it more sample efficient to train the discriminator.

[1] The same concept is typically called *experience replay* in reinforcement learning literature.

Specifically, we keep the generator and discriminator from ELECTRA, but add a memory buffer between them. In ELECTRA, the generator produces corrupted text examples by trying to recover masked tokens, and discriminator takes the corrupted text examples as input and is trained to predict if each token is either original or replaced. Because the training objective of the generator is to mimic the original input sentences based on the masked ones, the generator will gradually drift away from its original purpose of providing random replacements to the input. Thus, the generator has the inevitable trend to produce lower and lower quality sentences as the generator is being optimized. We use memory replay to alleviate the above issue, which we refer to as *distribution drift* in this paper. Our intuition is that, by saving examples produced by generator in the memory buffer, discriminator can reuse high-quality examples from the past, making it less sensitive to the distribution drift issue, and thus more sample efficient. We call this new transformer model *Transformer with Memory Replay* (TMR). Its architecture is illustrated in Figure 1. The detailed design of the memory replay would affect the performance of our new model. We will dicuss some design choices that lead to noticeable improvement on sample efficiency. Going beyond the design choices discussed in this paper, we hope that our new model can be viewed as a general framework that provides an easy way to manipulate training examples via various memory replay designs, thus possibly achieving even better sample efficiency.

Related Work

Transformer-based Models. Before the transformer architecture came along, the dominant sequence models for NLP were based on complex recurrent neural networks, which are very hard to parallelize. In (Vaswani et al. 2017), the *transformer* architecture, solely based on attention mechanisms and easily parallelizable, was introduced to the NLP community, which has superior performance than traditional recurrrent neural networks. Ever since, transformer-based models have been the top performers in various NLP task competitions. For example, BERT (Devlin et al. 2018) pretrains a large transformer on unlabeled text corpora using the masked-language modeling task, achieving significant improvement on GLUE and SQuAD benchmark datasets. This demonstrates the power of combining transformers and the self-supervised pre-training strategy. MASS (Song et al. 2019) and UniLM (Dong et al. 2019) extend BERT to the generation task by adding auto-regressive generative training objectives. Instead of masking out input tokens as in a masked-language modeling task for pre-training, XL-Net (Yang et al. 2019) masks attention weights such that the input sequence is autoregressively generated in a random order. ELECTRA corrupts the input by replacing some tokens with plausible alternatives sampled from a generator, which they showed is more sample efficient. It is also worth noting that there is significant effort dedicated to improving the transformer architecture to achieve computation efficiency (Kitaev, Kaiser, and Levskaya 2020; Kim and Awadalla 2020; Tay et al. 2020a; Rae et al. 2019; Tay et al. 2020b) or to extend beyond the NLP domain (Huang et al. 2018; Parmar et al. 2018; Karpov, Godin, and Tetko 2019; Girdhar et al. 2019; Huang and Yang 2020).

Memory Replay. Memory replay (or experience replay) is critical to deep reinforcement learning to achieve superhuman performance (Lin 1992; Schaul et al. 2015; Mnih et al. 2015). It has been shown to improve sample efficiency and stability by storing and reusing past transitions (Fedus et al. 2020). Some follow-up works have refined the basic memory replay mechanism (Schaul et al. 2015) in various ways (Horgan et al. 2018; Andrychowicz et al. 2017; Sun, Zhou, and Li 2020; Luo and Li 2020; Liu et al. 2019a). Several works have been trying to understand how memory replay works in the context of reinforcement learning. Liu and Zou (2018) study the effects of replay buffer size and mini-batch size on learning performance. It has been reported that agent performance is sensitive to the number of environment steps taken per gradient step (Fu et al. 2019). Sample efficiency can be improved by varying this ratio in combination with batch sizes (van Hasselt, Hessel, and Aslanides 2019). In addition to reinforcement learning, Wu et al. (2018) applies memory replay to GANs training in the task of learning new categories in a sequential fashion. They show that memory replay can prevent catastrophic forgetting (McCloskey and Cohen 1989), which is typically an issue in sequential learning.

Transformer with Memory Replay

Model Architecture

In this section, we describe the basic architecture of *Transformer with Memory Replay* (TMR). We keep the generator G and discriminator D from ELECTRA, but add a memory buffer between them. We describe each of them and explain the distribution drift issue which is the reason for using a memory buffer as follows.

Generator. The generator G is a transformer network (Vaswani et al. 2017) that maps a sequence of input tokens $\boldsymbol{x}^G = [x_1^G, \cdots, x_n^G]$ into a sequence of contextualized vector representations $h^G(\boldsymbol{x}^G) = [h_1^G, \cdots, h_n^G]$. The input token sequence is obtained from the original text token sequence $\boldsymbol{x} = [x_1, \cdots, x_n]$ by masking out a random set of positions, i.e., replacing the original token with the `[MASK]` token. Typically, 15% of input tokens are masked out randomly (Devlin et al. 2018; Clark et al. 2020). For any position t where the corresponding input token x_t^G is a `[MASK]` token, the generator outputs a probability for a particular token x with a softmax layer given by

$$p_G(x|\boldsymbol{x}^G) = \frac{\exp\left(e(x)^T h_t^G\right)}{\sum_{x'} \exp\left(e(x')^T h_t^G\right)} \tag{1}$$

where $e(x)$ is the embedding vector for token x. The generator is trained with the masked language modeling (MLM) task, i.e., it learns to predict the original tokens for the masked out positions. Denote the set of masked out positions as $\boldsymbol{m} = [m_1, \cdots, m_r]$. The loss function for the generator

is

$$L_G(\boldsymbol{x}^G, \theta_G) = \mathbb{E}\left(\sum_{i \in \boldsymbol{m}} -\log p_G(x_i^G | \boldsymbol{x}^G)\right). \quad (2)$$

Discriminator. Similar to the generator, the discriminator G is also a transformer network mapping a sequence of input tokens $\boldsymbol{x}^M = [x_1^M, \cdots, x_n^M]$ into a sequence of contextualized vector representations $h^D(\boldsymbol{x}^M) = [h_1^D, \cdots, h_n^D]$. Although the generator typically has the same model architecture as the discriminator, but smaller in size, the parameter values from the discriminator are used to initialize the task-specific model during the fine-tuning stage. Existing work assumes the input to the discriminator comes directly from the output of the generator via the inference process, i.e., replace each `[MASK]` token by a token that is sampled according to $p_G(\cdot|\boldsymbol{x}^G)$ (Clark et al. 2020). Instead, we use corrupted examples $\boldsymbol{x}^M$ sampled from the memory buffer as the input to the discriminator, in order to mitigate the distribution drift issue which we will elaborate shortly. For any position $t \in [1, n]$, the discriminator learns to predict whether the token x_t^M is *original* or *replaced*. A token is *original* if it matches the token id from the original text input. Otherwise, a token is considered *replaced*. The probability that token x_t^M is original is output by a sigmoid layer:

$$D(\boldsymbol{x}^M, t) = \text{sigmoid}\left(w^T h_t^D\right) \quad (3)$$

where w is the learnable parameter to the sigmoid layer. The loss function of discriminator is

$$\begin{aligned} L_D(\boldsymbol{x}^M, \theta_D) = &\mathbb{E}\left(\sum_{t=1}^{n} -\mathbb{1}(x_t^M = x_t) \log D(\boldsymbol{x}^M, t)\right. \\ &\left. -\mathbb{1}(x_t^M \neq x_t) \log(1 - D(\boldsymbol{x}^M, t))\right). \end{aligned} \quad (4)$$

Distribution Drift. We would like to point out the distribution drift issue of the generator, which is the key reason that motivates us to use the memory buffer. Specifically, to minimize the negative maximum likelihood loss L_G in Equation 2, the generator will tend to mimic the original input sentences and generate examples that highly resemble the input, as the training is making more and more progress. This behavior will cause the generator to drift away from its original purpose of providing random replacements to the input. Imagine an extreme scenario where the generator is highly optimized based on Equation 2, thus becoming perfectly capable of predicting the original token at each masked-out position. This generator would produce the highly similar (if not exactly the same) sentence as the original one. For example, if the original sentence $\boldsymbol{x}$ is "The individual images in a film are called frames" with tokens "image" and "film" being masked out, the sentence produced by a highly optimized generator $\boldsymbol{x}'^M$ would probably be "The individual images in a movie are called frames". When this sentence $\boldsymbol{x}'^M$ is received by the discriminator, all the tokens are considered as *original* except for the token "movie", because "movie" is the only token that is different from its corresponding one from the original sentence $\boldsymbol{x}$. The discriminator is supposed to learn reasonable language semantics by solving a two-class optimization problem based on these labeled tokens. However, there are two issues with $\boldsymbol{x}'^M$ that hinder the discriminator's learning progress:

1. the number of *replaced* tokens is much less than the number of `[MASK]` tokens (e.g., 50% less here), resulting in insufficient number of *replaced* tokens that would cause class imbalance problem for the discriminator;
2. the token "movie" that is considered as *replaced* is essentially noisy for the discriminator, because the sentence $\boldsymbol{x}'^M$ itself is completely acceptable, and it would be more reasonable to consider "movie" as *original*.

The above example shows how a low-quality sentence can hurt the discriminator's learning progress.

On the other hand, an insufficiently optimized generator would probably produce an unacceptable sentence, e.g., "The individual coma in a version are called frames". The tokens "coma" and "version" are considered as *replaced* while other tokens are *original*. This is a better-quality sentence, because these labeled tokens imply that "coma" and "version" are not semantically related to other tokens from this sentence. Unfortunately, the generator has the inevitable trend to produce lower and lower quality sentences as it is being optimized. We call this phenomenon the *distribution drift* of the generator. Memory replay is a mechanism that remembers and reuses past examples by saving to and replaying from a memory buffer. It can be used to alleviate the distribution drift issue by replaying high-quality sentences from the memory buffer as input to the discriminator. It is worth noting that memory replay can also be treated as an importance sampling technique (Zhao and Zhang 2015; Katharopoulos and Fleuret 2018), if we directly use it to hold the training set. This would enable other transformer models such as BERT to make use of memory replay. We do not consider this usage in this paper because memory replay is especially effective in handling dynamic example stream rather than static example set (Lin 1992; Schaul et al. 2015). Next we discuss the memory buffer that is inserted between the generator and the discriminator.

Memory Buffer. From p_G, we create a corrupted example $\boldsymbol{x}^M$ by replacing the `[MASK]` token in $\boldsymbol{x}$ with a token randomly sampled according to $p_G(\cdot|\boldsymbol{x}^G)$ for all positions in $\boldsymbol{m}$. Any corrupted example $\boldsymbol{x}^M$ will be saved into the memory buffer. We assign a real-valued weight to each example in the memory buffer, which indicates the importance of each example. While the example importance is not directly accessible, we use different strategies to approximate it, which will be discussed in the next subsection. The memory buffer has a fixed size N. To be scalable when N is large, we use a sum tree to maintain the example weights. The memory buffer should support at least three operators: `add`, `update` and `sample`. The operator `add` is used to add a new corrupted example $\boldsymbol{x}^M$ into the memory buffer. The operator `update` is used when we want to update the weight for an example already in the memory buffer. The operator `sample` is used to sample some examples from the memory buffer according to the weight distribution, i.e., the probability that any example is sampled is proportional to its weight. Any of these three operators has $O(\log N)$ in com-

plexity, because of the need to traverse the sum tree from the root to a leaf node. The operator `sample` mentioned here is essentially a stochastic sampling method. In general, the probability of sampling example i is

$$P(i) = \frac{w_i^\alpha}{\sum_j w_j^\alpha} \tag{5}$$

where w_j is the weight for example j and the hyperparameter α determines how much prioritization is used. For example, $\alpha = 0$ corresponds to uniform sampling, and $\alpha = \infty$ corresponds to greedy sampling (i.e., sample examples with the largest weights). We illustrate more on how memory buffer works with some concrete examples in Figure 2.

Joint Learning

We learn the generator and discriminator jointly by minimizing the combined loss

$$\min_{\theta_G, \theta_D} \sum_{\boldsymbol{x}^G, \boldsymbol{x}^M} L_G(\boldsymbol{x}^G, \theta_G) + \lambda L_D(\boldsymbol{x}^M, \theta_D) \tag{6}$$

where λ is a scalar that balances the above two loss terms. We use the same optimizer, Adam with warmup, as in Clark et al. (2020) to iteratively minimize the combined loss. Assume the mini-batch size is K.

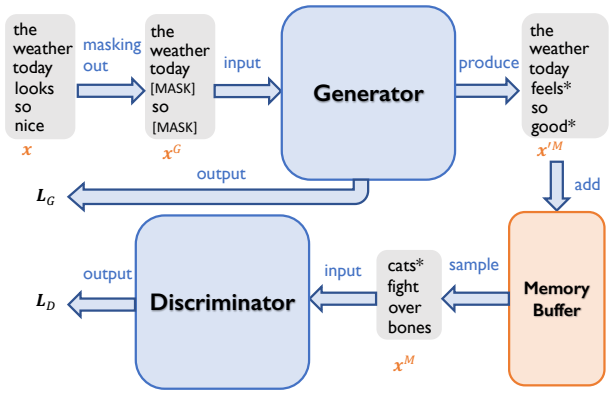


Figure 2: Forward pass of *Transformer with Memory Replay* (TMR). For simplicity, mini-batch size is set to 1 in the diagram to illustrate how an example is changed. We put $*$ right after a token to indicate this token has been replaced by the generator. Memory replay between the generator and discriminator helps the discriminator to learn more efficiently, by providing high-quality examples to it. The sentence "the weather today feels* so good*" is saved to the memory buffer, and a better-quality sentence "cats* fight over bones" which is sampled from the memory buffer is instead provided to the discriminator as input.

Forward Pass. Specifically, during the forward pass of each iteration, we sample a mini-batch of original text token sequence $\{\boldsymbol{x}_{(k)}\}_{k=1}^K$. For each $\boldsymbol{x}_{(k)}$, a random set of positions is selected and the corresponding tokens are replaced by the `[MASK]` token. Thus, we get $\{\boldsymbol{x}_{(k)}^G\}_{k=1}^K$ by masking out random tokens from $\{\boldsymbol{x}_{(k)}\}_{k=1}^K$. $\{\boldsymbol{x}_{(k)}^G\}_{k=1}^K$ are provided to the generator as input, and generator loss $L_G(\boldsymbol{x}^G, \theta_G)$ is computed according to Equation 2. In the meantime, from the generator output $p_G(x|\boldsymbol{x}^G)$, as in Equation 1, corrupted examples $\{\boldsymbol{x'}_{(k)}^M\}_{k=1}^K$ are created by replacing the `[MASK]` token in $\boldsymbol{x}$ with a token randomly sampled according to $p_G(\cdot|\boldsymbol{x}^G)$. The corrupted examples $\{\boldsymbol{x'}_{(k)}^M\}_{k=1}^K$ are saved into the memory buffer once created. To get the input for the discriminator, we sample a mini-batch of examples $\{\boldsymbol{x}_{(k)}^M\}_{k=1}^K$ from the memory buffer using the `sample` operator. The detailed design of the memory buffer (e.g., how to assign/update example weights) will affect which examples get sampled. Note that the sampled examples $\{\boldsymbol{x}_{(k)}^M\}_{k=1}^K$ are in general quite different from the examples $\{\boldsymbol{x'}_{(k)}^M\}_{k=1}^K$ that are just being saved to the memory buffer by the generator. The discriminator then computes the loss $L_D(\boldsymbol{x}^M, \theta_D)$ according to Equation 4. The forward pass is illustrated in Figure 2 with mini-batch size as 1.

Backward Pass. During the backward pass, gradient with respect to generator (or discriminator) parameters is computed from the generator (or discriminator) loss. As with (Clark et al. 2020), we don't back-propagate the discriminator loss through the generator, which is difficult because of the sampling operation. Adam with warmup is used to update the generator (or discriminator) parameters based on the generator (or discriminator) gradient.

Memory Replay Weights

Example weights play an important role in the memory replay mechanism. At each iteration, the generator produces a mini-batch of new corrupted examples $\{\boldsymbol{x'}_{(k)}^M\}_{k=1}^K$. The operator `add` is used to add these new examples into the memory buffer. When a new example is being added to the memory buffer, an initial weight is assigned to it. We then use the `sample` operator to sample a mini-batch of examples $\{\boldsymbol{x}_{(k)}^M\}_{k=1}^K$ from the memory buffer and provide them to the discriminator as input. The probability that an example is sampled is proportional to its weight. After forward and backward pass, we use the `update` operator to update the weights of the sampled examples based on feedback information from the discriminator. We hope to increase the weights of high-quality examples, because they are more beneficial to training the discriminator. In this way, these high-quality examples will be more likely to be sampled.

The strategy on how to assign and update example weights has great effect on which examples will be sampled as input to the discriminator, thus affecting its sample efficiency. For each new example added to the memory, we assign its initial weight as the current average of the weights. We also have tried other strategies to assign the initial weight, which are discussed in the experiments. Since the memory buffer has a fixed size, it will eventually become full as we add more and more examples to it. When we attempt to add a new example to a full memory, we evict the example with the lowest weight from the memory before adding the new example. The weight of each example indicates the importance of each example, which might change as the discriminator is being trained. To keep the example weight as up-to-date as possible, we update the weight of

each sampled example at the end of each iteration. We discuss two different strategies on updating example weights.

Loss Difference. We keep track of the discriminator loss L_D, as in Equation 4 for each example every time it is sampled. The loss difference of the current time and the previous time is used as the new weight. Note that, since computing the loss difference requires that an example gets sampled at least twice, we will not update the weight for any new example until the second time it is sampled. In this case, we do not simply use the loss value as the new weight, because the loss value typically decreases each time an example is sampled, so a smaller weight would not necessarily imply that it is less important. Instead, it typically means that this example has been recently sampled. Our experiments also show that using loss value as the weight results in bad performance.

Gradient Norm. Gradient norm has been shown to be the optimal weight for each example in importance sampling (Zhao and Zhang 2015). It demonstrates that per-example gradient norm can be a strong indicator for the example's importance. Each time an example is sampled, we update its weight using the discriminator gradient norm for this example. The drawback of using per-example gradient norm is that it incurs some overhead to compute the per-example gradient for each example in the mini-batch (Goodfellow 2015). Recently, Katharopoulos and Fleuret (2018) proposed an upper bound on gradient norm that can be computed efficiently. This upper bound has been shown to be a reasonable approximation on the gradient norm in training various neural network models (Liu, Wu, and Mozafari 2020). We also use this upper bound as the new weight.

Experimental Setup

Pre-training

We pre-train our model with two different sizes: a small model and a base model on English Wikipedia. The detailed hyperparameter values are included in the appendix. As suggested by Clark et al. (2020), in addition to sharing the embedding table between input and output tokens for the generator, we also share the embedding table beween generator and discriminator. We use Adam with warmup to pre-train the models. The detailed setup is the same as Clark et al. (2020) if not stated otherwise. Specifically, we set $\epsilon = 1e-6$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The mini-batch size is 128 for the small model and 256 for the base model. The memory buffer size N is set to 1k in our experiments.

Fine-tuning

We use two commonly used datasets as the benchmark to evaluate performance: General Language Understanding Evaluation (GLUE) (Wang et al. 2018) and Stanford Question Answering Dataset (SQuAD) (Rajpurkar et al. 2016).

General Language Understanding Evaluation (GLUE). GLUE consists of eight tasks (i.e., MNLI, QQP, QNLI, SST, CoLA, STS, MRPC and RTE)[2], each of which corresponds to a specific type of NLP problem. As with Clark et al. (2020), our evaluation metrics are Spearman correlation for STS, Matthews Correlation for CoLA, and accuracy for other GLUE tasks. The average of these scores is reported. Unless stated otherwise, results are on the dev set. For fine-tuning, we only need the discriminator's parameters to initialize task-specific models (Clark et al. 2020). Specifically, we use the final hidden vector $h_1^D \in \mathbb{R}^H$ of the discriminator corresponding to the first input token (`[CLS]`) as the aggregate representation (Devlin et al. 2018). Note that H is the hidden size. The only new parameters introduced during fine-tuning are classification layer weights $W \in \mathbb{R}^{C \times H}$, where C is the number of labels. We use the standard cross-entropy loss for classification tasks: $-\log\left(\text{softmax}\left(h_1^D W^T\right)_{[class]}\right)$ where $[class]$ is the true class index.

Stanford Question Answering Dataset (SQuAD). The Stanford Question Answering Dataset (SQuAD v1.1) is a collection of 100k crowd-sourced question/answer pairs (Rajpurkar et al. 2016). Given a question and a passage from Wikipedia containing the answer, the task of SQuAD is to predict the answer text span in the passage. As with the GLUE benchmark, for fine-tuning, we only need the discriminator's parameters to initialize task-specific models (Clark et al. 2020). We introduce a start vector $S \in \mathbb{R}^H$ and an end vector $E \in \mathbb{R}^H$ to the task-specific model. The probability of location i being the start of the answer span is computed as a dot product between h_i^D and S, followed by a softmax over all locations in the paragraph:

$$P_i^S = \frac{e^{S \cdot h_i^D}}{\sum_j e^{S \cdot h_j^D}}. \tag{7}$$

Similarly, the probability of location i being the end of the answer span is computed as a dot product between h_i^D and E, followed by a softmax over all locations in the paragraph:

$$P_i^E = \frac{e^{E \cdot h_i^D}}{\sum_j e^{E \cdot h_j^D}}. \tag{8}$$

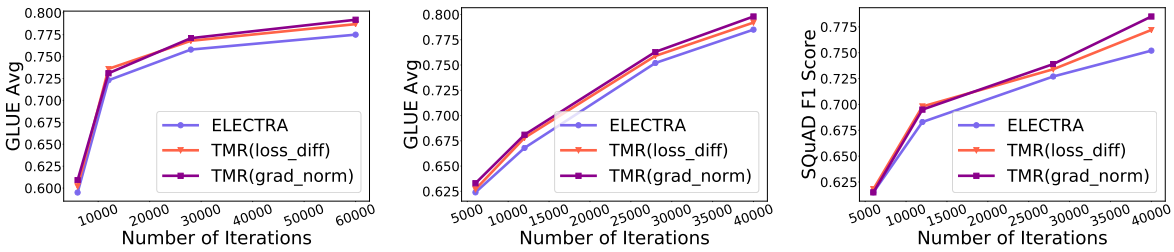
The loss for fine-tuning is $-\log P^S_{[start]} - \log P^E_{[end]}$, where $[start]$ and $[end]$ are the correct start and end positions. The score of a candidate span from position i to position j is defined as $S \cdot h_i^D + E \cdot h_j^D$. The span with highest score where $i \leq j$ is used as the prediction.

Experimental Results

Pre-training Efficiency

To show that memory replay can improve the sample efficiency, we compare our models (i.e., TMR(loss_diff) and TMR(grad_norm)) with baseline ELECTRA after being pre-trained with the same number of iterations (i.e., also the same number of examples, because we use the same mini-batch size for different methods). TMR(loss_diff) and TMR(grad_norm) are our model *Transformer with Memory Replay* using loss difference and gradient norm to update example weight, respectively. Specifically, we pre-train different models to the same number of iterations using the same setting. We then fine-tune the models on GLUE and SQuAD

[2]It is customary to exclude WNLI because it is difficult to beat even the majority classifier.



(a) Small model on GLUE (b) Base model on GLUE (c) Base model on SQuAD

Figure 3: Pre-training efficiency comparison. Our models (i.e., TMR(loss_diff) and TMR(grad_norm)) have better scores than baseline ELECTRA when the pre-training has gone through enough iterations, because the memory replay needs time to accumulate enough high-quality examples.

Table 1: Results on GLUE of the small models after pre-trained for 200k steps.

Model	MNLI	QQP	QNLI	SST	CoLA	STS	MRPC	RTE	Avg.
ELECTRA	0.730	0.881	0.836	0.862	0.733	0.832	0.809	0.584	0.783
TMR(loss_diff)	0.753	0.887	0.846	0.868	0.737	0.835	0.811	0.613	0.794

to compare their performance. The results are illustrated in Figure 3. When the number of iterations is small, our models have almost the same performance on both GLUE and SQuAD benchmarks. As the number of iterations increases, we can see an obvious gap between our models and baseline. This is because, as more and more examples are processed, our models utilize the memory replay to remember more and more high-quality examples, thus boosting the discriminator's learning speed. On the other hand, the baseline ELECTRA uses whatever examples that are produced by the generator to train the discriminator. As the training of the generator continues, the distribution of examples produced will also change due to the distribution drift issue, affecting the discriminator's training. We list the detailed results for each GLUE task after 200k steps in Table 1. As expected, after the same number of pre-training steps, our models TMR have better performance than baseline across most of the GLUE tasks. Note that in the original paper (Clark et al. 2020), the baseline ELECTRA is trained with much more iterations, getting slightly better GLUE scores than reported here. We do not train the models for too many iterations because the max number of iterations used in our experiments suffices to demonstrate the benefits of the proposed memory buffer mechanism (especially given the increasing concern on the effect of excessive energy consumption on the environment (Strubell, Ganesh, and McCallum 2019; Schwartz et al. 2019)).

We also observe that TMR(grad_norm) often has better results than TMR(loss_diff). Theoretical analysis has been provided to show that gradient norm is the optimal weight for each example in importance sampling (Zhao and Zhang 2015). Echoing this theoretical point, our observation empirically demonstrates that gradient norm is also a good choice for example weight in memory replay. Loss difference can be a bad approximation to gradient norm because model parameters might have changed significantly between adjacent visits to the same example. However, loss difference is a relatively cheap way to measure the example importance, which is especially beneficial in terms of the runtime efficiency as discussed right below.

Runtime Efficiency. We also report the runtime efficiency in terms of the wall-clock time needed for training. As shown in Figure 4, using loss difference to measure the example importance reduces the wall-clock time overhead of the memory replay, leading to better runtime efficiency.

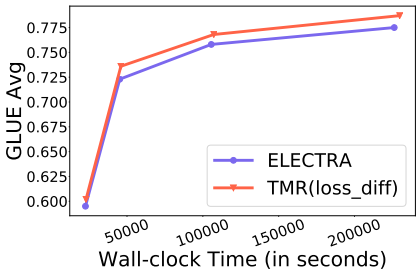


Figure 4: Runtime efficiency comparison using small model on GLUE. TMR(loss_diff) has better runtime efficiency than the baseline ELECTRA, because TMR(loss_diff) achieves better scores after pretaining for the same amount of wall-clock time.

Alternative Strategies for Weight Initialization

We discuss some alternative strategies that we have tried but have worse performance. We initialize the weight of a new example using the average of example weights in our previous experiments. This strategy will assign the same weight to new examples that arrive at the same iteration (i.e., these new examples are in the same mini-batch). The difference among examples is essentially ignored by this strategy. Intuitively, it might be better to assign weight to a new example using the weight of a similar example that is already present in the memory buffer. Since each example is represented by a sequence of token indices $\boldsymbol{x}^M = [x_1^M, x_2^M, \cdots, x_n^M]$, which can be viewed as the feature vector of this example, the similarity of two examples can be measured by how similar their features are. There are two alternative strategies that take into consideration the similarity among exam-

ples: *Least Square Regression* and *Linear Upper Confidence Bound*. Before we dive into these two strategies, let us assume the memory buffer contains a set of examples and their corresponding weights, denoted as $\{\boldsymbol{x}^B_{(i)}, r_i\}_{i=1}^N$, where r_i is the weight of the example $\boldsymbol{x}^B_{(i)}$ and N is the capacity of the memory buffer.

Least Square Regression. To determine the weights for a new example, we solve a *Least Square Regression problem* (LSR) based on examples in the memory buffer and their weights:

$$\theta^* = arg\,\min_{\theta} \sum_{i=1}^{N} (\theta^T \boldsymbol{x}^B_{(i)} - r_i)^2. \tag{9}$$

Then, we assign ${\theta^*}^T \boldsymbol{x}^M$ as the initial weight to the new example $\boldsymbol{x}^M$. Examples with different features will get different initial weights.

Linear Upper Confidence Bound. Multi-armed bandit problem (Bubeck, Munos, and Stoltz 2009; Even-Dar, Mannor, and Mansour 2002) provides a general framework for studying efficient sampling methods. For example, it has been used to speed up optimization methods for model training (Salehi, Thiran, and Celis 2017; Liu, Wu, and Mozafari 2020), maximum inner product search (Liu, Wu, and Mozafari 2019), hyperparameter search (Li et al. 2017), and model-related query execution (He et al. 2020). We cast the memory replay into a multi-armed bandit problem, and rely on a bandit method to initialize and update the example weights. Specifically, we treat each example as an arm. In this way, sampling an example is equivalent to picking an arm to pull, and a high-quality example corresponds to an arm with high reward. Because each example is represented as its feature vector, there are an infinite number of possible examples, implying an infinite number of arms. To deal with an infinite number of arms, we resort to linear bandit, a special bandit setting where each arm is represented by a feature vector and reward is assumed to have a linear relationship with the arm feature vector. We use a popular linear bandit method called *Linear Upper Confidence Bound* (LinUCB), as described in Algorithm 1 of Chu et al. (2011). The Least Square Regression strategy will not change the inital weight of a new example until it gets sampled. LinUCB can essentially keep refining initial weights of new examples as more and more examples are sampled.

We did some experiments using LSR and LinUCB. The results are shown in Figure 5. We can see that both TMR(LSR) and TMR(LinUCB) are much worse than TMR(loss_diff). These results are a bit counter-intuitive, because the idea of assigning weight to a new example using the weight of a similar example is reasonable. We suspect the reason why neither TMR(LSR) nor TMR(LinUCB) work is that the linear relationship between feature vector and weight/reward, which is assumed by both, does not hold in our memory buffer. It would be a good future work direction to consider weight initialization strategies without linear relationship assumption.

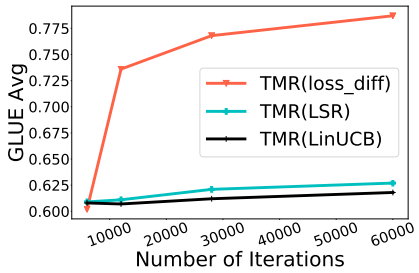


Figure 5: Comparison of alternative strategies for weight initialization. TMR(LSR) and TMR(LinUCB) are much worse than TMR(loss_diff) as the number of iterations increases.

Table 2: Runtime cost comparison. The value in the table is the wall clock time (in seconds) to finish 100 iterations.

Model	Time	Model	Time
ELECTRA	377	TMR(grad_norm)	937
TMR(loss_diff)	383	TMR(grad_bound)	432

Runtime Cost

We investigate the runtime cost for different models. In Table 2, we list the wall clock time to finish 100 iterations of training small models. The baseline ELECTRA model has the lowest time cost, which is expected, because our models have the overhead of maintaining the memory buffer. However, the time cost of TMR(loss_diff) is only slightly larger than that of ELECTRA. This implies that the overhead of maintaining the memory buffer can be very small, because computing loss difference will not incur additional cost. On the other hand, the time cost of TMR(grad_norm) is way larger than the baseline. This is because accurately computing per-example gradient norm requires us to feed each example of a mini-batch separately to the model for both forward and backward pass. This is usually very costly, because high parallelism provided by GPU cannot be fully utilized. TMR(grad_bound) uses the gradient upper bound from Katharopoulos and Fleuret (2018) as an approximation to gradient norm. It can significantly reduce the time cost of TMR(grad_norm).

Conclusion

We have proposed a new transformer model, called *Transformer with Memory Replay* (TMR), for improving the sample efficiency. Our model integrates a memory replay mechanism into ELECTRA by adding a memory buffer between the generator and discriminator. Because the memory replay mechanism enables examples produced by the generator to be reused multiple times, the discriminator can be trained with high-quality examples, making it more sample efficient. We also introduced two different strategies on how to update example weights (i.e., use loss difference or gradient norm). Our experiments have shown that our models achieve higher scores than baseline when pre-trained for the same number of iterations (i.e., using the same amount of examples). We further demonstrated that the loss difference strategy leads to better runtime efficiency because of its low wall-clock time overhead.

Acknowledgments

This work is in part supported by National Science Foundation and gifts from Google. The authors would like to thank anonymous reviewers for their insightful feedback and suggestions.

References

Andrychowicz, M.; Wolski, F.; Ray, A.; Schneider, J.; Fong, R.; Welinder, P.; McGrew, B.; Tobin, J.; Abbeel, P.; and Zaremba, W. 2017. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Bubeck, S.; Munos, R.; and Stoltz, G. 2009. Pure exploration in multi-armed bandits problems. In *International conference on Algorithmic learning theory*, 23–37. Springer.

Chu, W.; Li, L.; Reyzin, L.; and Schapire, R. 2011. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 208–214. JMLR Workshop and Conference Proceedings.

Clark, K.; Luong, M.-T.; Le, Q. V.; and Manning, C. D. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J.; Le, Q. V.; and Salakhutdinov, R. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Dong, L.; Yang, N.; Wang, W.; Wei, F.; Liu, X.; Wang, Y.; Gao, J.; Zhou, M.; and Hon, H.-W. 2019. Unified language model pre-training for natural language understanding and generation. *arXiv preprint arXiv:1905.03197*.

Even-Dar, E.; Mannor, S.; and Mansour, Y. 2002. PAC bounds for multi-armed bandit and Markov decision processes. In *International Conference on Computational Learning Theory*, 255–270. Springer.

Fedus, W.; Ramachandran, P.; Agarwal, R.; Bengio, Y.; Larochelle, H.; Rowland, M.; and Dabney, W. 2020. Revisiting fundamentals of experience replay. In *International Conference on Machine Learning*, 3061–3071. PMLR.

Fu, J.; Kumar, A.; Soh, M.; and Levine, S. 2019. Diagnosing bottlenecks in deep q-learning algorithms. In *International Conference on Machine Learning*, 2021–2030. PMLR.

Girdhar, R.; Carreira, J.; Doersch, C.; and Zisserman, A. 2019. Video action transformer network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 244–253.

Goodfellow, I. 2015. Efficient per-example gradient computations. *arXiv preprint arXiv:1510.01799*.

He, W.; Anderson, M. R.; Strome, M.; and Cafarella, M. 2020. A Method for Optimizing Opaque Filter Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 1257–1272.

Horgan, D.; Quan, J.; Budden, D.; Barth-Maron, G.; Hessel, M.; Van Hasselt, H.; and Silver, D. 2018. Distributed prioritized experience replay. *arXiv preprint arXiv:1803.00933*.

Huang, C.-Z. A.; Vaswani, A.; Uszkoreit, J.; Shazeer, N.; Simon, I.; Hawthorne, C.; Dai, A. M.; Hoffman, M. D.; Dinculescu, M.; and Eck, D. 2018. Music transformer. *arXiv preprint arXiv:1809.04281*.

Huang, Y.-S.; and Yang, Y.-H. 2020. Pop Music Transformer: Beat-based modeling and generation of expressive Pop piano compositions. In *Proceedings of the 28th ACM International Conference on Multimedia*, 1180–1188.

Karpov, P.; Godin, G.; and Tetko, I. V. 2019. A transformer model for retrosynthesis. In *International Conference on Artificial Neural Networks*, 817–830. Springer.

Katharopoulos, A.; and Fleuret, F. 2018. Not all samples are created equal: Deep learning with importance sampling. In *International conference on machine learning*, 2525–2534. PMLR.

Kim, Y. J.; and Awadalla, H. H. 2020. Fastformers: Highly efficient transformer models for natural language understanding. *arXiv preprint arXiv:2010.13382*.

Kitaev, N.; Kaiser, Ł.; and Levskaya, A. 2020. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*.

Li, L.; Jamieson, K.; DeSalvo, G.; Rostamizadeh, A.; and Talwalkar, A. 2017. Hyperband: A novel bandit-based approach to hyperparameter optimization. *The Journal of Machine Learning Research*, 18(1): 6765–6816.

Lin, L.-J. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4): 293–321.

Liu, H.; Trott, A.; Socher, R.; and Xiong, C. 2019a. Competitive experience replay. *arXiv preprint arXiv:1902.00528*.

Liu, R.; Wu, T.; and Mozafari, B. 2019. A bandit approach to maximum inner product search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4376–4383.

Liu, R.; Wu, T.; and Mozafari, B. 2020. Adam with Bandit Sampling for Deep Learning. *arXiv preprint arXiv:2010.12986*.

Liu, R.; and Zou, J. 2018. The effects of memory replay in reinforcement learning. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 478–485. IEEE.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019b. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Luo, J.; and Li, H. 2020. Dynamic experience replay. In *Conference on Robot Learning*, 1191–1200. PMLR.

McCloskey, M.; and Cohen, N. J. 1989. Catastrophic interference in connectionist networks: The sequential learning

problem. In *Psychology of learning and motivation*, volume 24, 109–165. Elsevier.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature*, 518(7540): 529–533.

Parmar, N.; Vaswani, A.; Uszkoreit, J.; Kaiser, L.; Shazeer, N.; Ku, A.; and Tran, D. 2018. Image transformer. In *International Conference on Machine Learning*, 4055–4064. PMLR.

Rae, J. W.; Potapenko, A.; Jayakumar, S. M.; and Lillicrap, T. P. 2019. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*.

Rajpurkar, P.; Zhang, J.; Lopyrev, K.; and Liang, P. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Salehi, F.; Thiran, P.; and Celis, L. E. 2017. Coordinate descent with bandit sampling. *arXiv preprint arXiv:1712.03010*.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2015. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Schwartz, R.; Dodge, J.; Smith, N. A.; and Etzioni, O. 2019. Green ai. *arXiv preprint arXiv:1907.10597*.

Song, K.; Tan, X.; Qin, T.; Lu, J.; and Liu, T.-Y. 2019. Mass: Masked sequence to sequence pre-training for language generation. *arXiv preprint arXiv:1905.02450*.

Strubell, E.; Ganesh, A.; and McCallum, A. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*.

Sun, P.; Zhou, W.; and Li, H. 2020. Attentive experience replay. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 5900–5907.

Tay, Y.; Bahri, D.; Yang, L.; Metzler, D.; and Juan, D.-C. 2020a. Sparse sinkhorn attention. In *International Conference on Machine Learning*, 9438–9447. PMLR.

Tay, Y.; Dehghani, M.; Bahri, D.; and Metzler, D. 2020b. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*.

van Hasselt, H.; Hessel, M.; and Aslanides, J. 2019. When to use parametric models in reinforcement learning? *arXiv preprint arXiv:1906.05243*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Wang, A.; Singh, A.; Michael, J.; Hill, F.; Levy, O.; and Bowman, S. R. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, Z.; Bapst, V.; Heess, N.; Mnih, V.; Munos, R.; Kavukcuoglu, K.; and de Freitas, N. 2016. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*.

Wu, C.; Herranz, L.; Liu, X.; van de Weijer, J.; Raducanu, B.; et al. 2018. Memory replay gans: Learning to generate new categories without forgetting. *Advances in Neural Information Processing Systems*, 31: 5962–5972.

Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J.; Salakhutdinov, R.; and Le, Q. V. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*.

Zhao, P.; and Zhang, T. 2015. Stochastic optimization with importance sampling for regularized loss minimization. In *international conference on machine learning*, 1–9. PMLR.