# Inquisition: A Reformed Question-Answering System

Stanford CS224N Default Project
**Track:** SQuAD
**TA Mentor:** Kaili

**Andrew Gaut**
Department of Computer Science
Stanford University
agaut@stanford.edu

**Shannon Yan**
Department of Computer Science
Stanford University
shannony@stanford.edu

**Zeb Mehring**
Department of Computer Science
Stanford University
zebm@stanford.edu

## Abstract

Transformer-based models are incredibly powerful, but are also incredibly expensive to train and utilize for inference at scale. In this project, we apply memory- and compute-reduction techniques (such using LSH attention with chunking and RevTransformer blocks) to a popular transformer-based NLP QA model (QANet) to explore whether more efficient models can achieve similar performance to the more expensive ones highlighted in most modern research papers. We have implemented the base QANet model from scratch, and will discuss the impact of replacing core subnetworks within the model with more resource-efficient implementations using the techniques described in the Reformer paper.

## 1 Introduction

In the field of NLP, recent work has shifted focus from recurrent architectures (like vanilla RNNs, LSTMs, and GRUs) [1] to self-attention-based "transformer" models, in part because self-attention encoder computations can be effectively parallelized on modern hardware while recurrent computations cannot be [2].

Transformer-based models have achieved state of the art performance on a number of tasks, but they also come with high memory and computational requirements (consider the quadratic complexity required to compute the matrix-matrix product involved in self-attention) [3] [4] [5], [6]. Question-answering is one such NLP task in which transformer-based models have been prominent in recent work. QANet, which makes use of a modified transformer encoder to produce embeddings, is a particularly powerful model for addressing this problem [7]. It is the subject of study of this project.

A number of papers have proposed modifications to the transformer's core architecture to cut the computational complexity of self-attention to linear or log-linear time and/or space complexity [8]. We experiment with one such architecture adjustment, the Reformer [9], in this project. Specifically, we apply modifications proposed in the Reformer paper to the transformer-based component of QANet to improve memory and computational complexity. Our contributions are summarized as follows:

- We implement QANet from scratch and achieve good performance on SQuAD 2.0.
- We find that LSH significantly lowers QANet performance on SQuAD 2.0.
- We find that applying LSH and applying LSH and RevTransformer to QANet decreases memory usage, but can increase training time.

## 2 Related Work

Because of the ubiquity of transformer-based architectures in machine learning research in recent years, significant research has been made into the development of efficiency-boosting architecture modifications [8]. Broadly speaking, some such methods attempt to use static or learned sparsity patterns in the matrix product $QK^\top$ (whose rows represent unnormalized attention scores). Examples of such approaches include the Sinkhorn Transformer, Routing Transformer, Sparse Transformer, and Reformer [10] [11] [12][9].

Prior work has investigated performance gains made by making architecture modifications to the transformer across tasks and applications and found that most modifications did not improve transformer performance across a wide variety of tasks [13]. To our knowledge, we are the first to examine Reformer performance on the question answering task and, in that sense, we are exploring the generalizability of reformer to the question answering task. Moreover, the performance losses resulting from using LSH (as discussed below) suggest that a study similar to Narang et al [13], in which efficiency improvements for transformer are evaluated robustly across tasks, may be necessary.

## 3 Approach

First, we evaluate the baseline BiDAF model on SQuAD. For details on the BiDAF architecture, see [14]. Then, we implement an existing QA model – QANet [7] – and apply the modifications suggested in the Reformer architecture to it [9].

Our first task was implementing a version of QANet **from scratch, with minimal references to existing code**. This non-trivial task required the creation of four major model components: an input embedding layer, QANet encoder blocks, a context-query attention layer, and the output layer. The overall model architecture can be seen in Figure 2.

Our second task was augmenting this base model with memory-saving improvements suggested by the Reformer paper, namely LSH self-attention and RevTransformer residual blocks. We integrated with an open-source library to provide this functionality. We then compared the efficiency and performance of the resulting modified model against the baseline we implemented in the previous task.

### 3.1 Model Components

**Input Embedding Layer:** The input embedding layer takes in word and character embeddings for a snippet of text and produces a single embedding to be used by the encoder embedding blocks. We use a two dimensional convolution on the character embeddings only with $d_{\text{model}}$ output channels and a kernel size of $(1, 5)$, and run it over our input of shape (batch size, character embedding size, sequence length, word length) and then max pool over the word length dimension. We use a kernel height of 5 (provided by the QANet authors) and a width of 1 to preserve the sequence length dimension. Later, we concatenate the convolved character embedding and the word embeddings and convolve that again using a one dimensional convolution with $d_{\text{model}}$ output channels and run the result through a two-layer highway encoder network.

**QANet Encoder Block:** QANet proposes the Encoder block, which is essentially a transformer encoder block with minor modifications. Namely, the encoder block repeatedly applies convolutions (and layer norms) to the input before applying the usual multihead self-attention and feed-forward neural network (along with residual connections and layer normalization between each layer). The intuition behind this architectural choice is that the convolutions will capture local text dependencies between words close together in the text, while self-attention focuses on global dependencies between word-pairs.

In exact terms, the Encoder block is constructed as follows: [1D Convolution $\times$ # + self-attention layer + feed-forward layer] and is shown in Figure 1.

Each of these operations (convolution/self-attention/feed-forward) is placed inside a residual block as shown in Figure 1. In mathematical terms, for an input $x$ and an operation $f$, the output of a residual block containing that operation is $f(\text{layernorm}(x)) + x$ where $\text{layernorm}$ indicates layer-normalization [15].

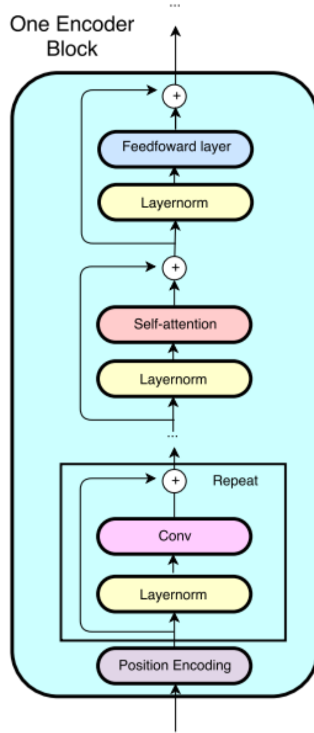


Figure 1: QANet encoder block architecture [7]. The input is convolved repeatedly before being fed through self-attention and a feed-forward output layer. There are residual connections and layer normalization layers between each step.

We use depthwise separable convolutions, as the authors of the Reformer paper observed that depthwise separable convolution was more memory efficient and generalized better [9][16]. Each encoder block contains 4 convolutional residual blocks.

The self-attention layer in each encoder block has 8 attention heads. In general, we can stack encoder blocks together, and each stacked embedding encoder block consists of 2 individual encoder blocks. Each block in the stack shares parameters with the other(s).

**QANet Encoder Block Modifications:** This section describes our novel contributions to the QANet architecture. We use three types of encoder blocks in our experiments: the vanilla encoder block described above, the LSH encoder block, and the LSH+RevTransformer encoder block.

The LSH encoder block uses LSH self-attention rather than standard, dot product self-attention. Intuitively, LSH self-attention leverages sparsity in the $QK^\top$ matrix to avoid unnecessary computation, since if $(QK^\top)_{ij}$ is small, it implies that attention score $\alpha_{ij}$ will be negligible after softmax is applied.[1] To begin, they use a locality-sensitive hashing (LSH) function, which takes vectors and maps them to buckets such that all vectors in a given bucket are nearby in the embedding space. Then, the idea is that the dot product between queries in different buckets will be small enough that their attention scores would be negligible. Queries in $Q$ are then *grouped* by their hash value, and self-attend *only* to queries within the same bucket. Softmax is applied to the reduced self-attention scores, and the results are recombined into an output matrix.

The Reformer also suggests Reversible Transformer modifications. In particular, embeddings $x$ of shape $(B, L, D)$ input to the Reformer are split along the last dimension into two embeddings, $x_1$ and $x_2$, both of shape $(B, L, D/2)$. Then, the layer output $[y_1, y_2]$ is computed as:

$$y_1 = x_1 + \text{LSHAttention}(x_2) \qquad y_2 = x_2 + \text{FeedForward}(y_1) \tag{1}$$

[1] The Reformer authors note that one can set $Q = K$ without losing performance. Our Reformer implementation does this as well.

We can then compute the activations for previous layers from subsequent layers since $x_2 = y_2 - \text{FeedForward}(y_1)$ and $x_1 = y_1 - \text{LSHAttention}(x_2)$, where $y_1$ and $y_2$ can be obtained from the current layer's outputs of shape $(B, L, D)$ by taking $y_1$ of shape $(B, L, D/2)$ to be the first $D/2$ embedding dimensions and $y_2$ to be the rest. We did not implement LSH and the RevTransformer modules from scratch, and instead adapted an open source implementation [17] to work with our QANet implementation.

**Context-query attention layer:** First, we compute the similarities between each pair of context and query tokens to create a similarity matrix $S \in \mathbb{R}^{n \times m}$ (where $n$ is the number of tokens in the context and $m$ is the number of tokens in the query). Then, we normalize each row of $S$ by applying the softmax function, getting a matrix $\tilde{S}$. Afterwards, we compute context-to-query attention as $A = \tilde{S} \cdot Q \in \mathbb{R}^{n \times d}$ where $Q$ is the encoded query (and is of shape $m \times d$, where $d$ is the encoding dimension). The similarity function used is the trilinear function: $f(q, c) = W_0[q, c, q \odot c]$ where $W_0$ is a trainable variable, $Q$ is the encoded query, and $C$ is the encoded context. For context-to-query attention, we compute the column normalized matrix $\bar{S}$ of $S$ by softmax function, and use this to compute query-to-context attention: $B = \tilde{S} \cdot \bar{S}^\top \cdot C$.

**Output Layer:** This final layer is relatively simple, as it re-uses the stacked encoder block module described above. The inputs to this layer are $M_0 = \text{StackedModelEncoder}(x)$, $M_1 = \text{StackedModelEncoder}(M_0)$, and $M_2 = \text{StackedModelEncoder}(M_1)$, where $x$ is the output from the context-to-query attention layer. The set of stacked encoder blocks used here have 7 constituent encoder blocks and share weights (as before). The output layer then computes the logits for the start of the answer as $\text{FeedForward}([M_0; M_1])$ and the logits for end of the answer as $\text{FeedForward}([M_0; M_2])$. We finally apply a softmax to obtain the log probabilities.

We used basic PyTorch building blocks (`nn.Linear`, `nn.LayerNorm`, `nn.Softmax`, etc.) to implement each of these components based on the description provided in the QANet paper [7]. We made use of provided code for loading the SQuAD dataset, obtaining input pretrained embeddings, and training the model. Since the conceptual architecture of the model is similar to BiDAF [14], we also made use of the provided BiDAF implementation to do "hotswap testing" (plugging in our own model components in lieu of existing components in the BiDAF model to ensure performance parity).

# 4 Experiments

## 4.1 Data

Our dataset was the publicly available SQuAD 2.0 [18] dataset, which contains 129,941 training examples, 6,078 dev examples, and 5,915 test examples used for question answering tasks. The examples come in the form of context paragraph-question pairs; for example, the model will be given a paragraph (the "context") from Wikipedia and a question regarding that paragraph. The model is expected to answer the question by selecting a subspan of tokens in the context paragraph (i.e. the correct answer is a substring within the context paragraph and it is the model's job to identify it).

## 4.2 Evaluation method

For evaluating performance, we use F1 and EM scores averaged over the entire evaluation dataset to compare models. To compare memory usage, we consider the amount of GPU memory allocated during model training. This is measured using Weights and Biases [19]. We also consider model runtime during training.

## 4.3 Experimental details

First, we discuss our hyperparameter settings. We use most of the hyperparameters specified by the QANet paper, including the number of convolutions per encoder block (4 and 2 for the embedding encoder and model encoder, respectively), the dropout probability (0.1), and $d_{\text{model}}$ (128). However, we make some adjustments. We use only 5 model encoder blocks in the stacked model encoder rather than the recommended 7 due to GPU RAM constraints.

Moreover, we elect to use Adadelta rather than the Adam optimizer used by QANet, and we use our own starting learning rate (though, as Adadelta is an adaptive algorithm, this choice is not very

Figure 2: QANet model architecture [7]. Each input (context and query) are embedded, encoded, and then mixed via an attention layer. The resulting representation is then encoded several more times, before final neural networks are applied to predict the start and end positions of the answer.

impactful). We do this because the QANet authors had access to more data on which to cross-validate their model and select an optimal Adam learning rate. Since Adadelta does not require cross-validation of the learning rate and since we had limited computing resources, we elected to use this more forgiving option. We use the same hyperparameters for vanilla QANet, LSH QANet (QANet modified both with LSH self-attention), and Reformer QANet (QANet modified both with LSH self-attention and reversible transformer blocks).

We fully trained and tested the vanilla QANet and LSH QANet on the SQuAD 2.0 dataset and submitted to the dev and test leaderboards to evaluate performance. Due to computational constraints, we were unable to complete a full training run for our Reformer model (approximately 10 hours of training, which exceeded the computational budget granted to us by the course after evaluating our other models), but ablation studies suggest its performance is on par with the LSH model.

For all three architecture variants, we evaluate memory and time usage per mini-batch. Because memory and time usage is relatively consistent between mini-batches, we monitor GPU usage and time usage for one mini-batch only.

### 4.4 Results

Performance results can be seen in Table 1. QANet outperforms the BiDAF baseline, as expected.

However, we see a sizable performance drop when we use LSH instead of standard dot product self-attention. This was not expected; transformers with Reformer modifications should maintain similar performance according to results from the Reformer paper. There are several possible explanations for this. First, we did not have a sufficient computational budget to properly tune hyperparameters associated with LSH (we used hyperparameters similar to those used by the original Reformer paper). However, different hyperparamters may be more suited to the SQuAD dataset, and usage of those hyperparameters may significantly boost performance. The second possible explanation is that the performance results reported in the Reformer paper do not generalize across tasks; perhaps the

Reformer modifications cause significant performance reduction when used in the QA setting. This could be suggestive to a phenomenon similar to that found in Narang et al [13] wherein transformer modifications for efficiency can maintain performance in a narrow set of tasks but do not generalize well to others.

| Model | EM Score | F1 Score |
|---|---|---|
| Baseline - BiDaF | 57.5 | 61.0 |
| LSH | 57.9 | 61.7 |
| QANet | 62.0 | 65.0 |

Table 1: EM and F1 scores of BiDaF, LSH, and QANet on the test set.

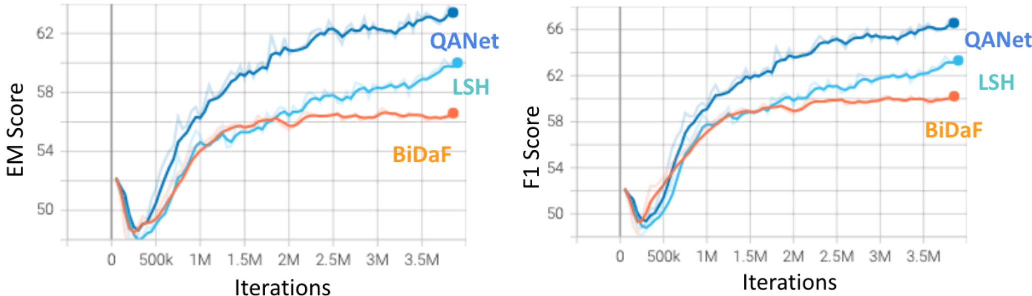


Figure 3: EM and F1 scores of QANet, LSH, and BiDaF on the dev set. While QANet outperforms BiDaF as expected, we find that LSH leads to a significant decrease in performance.

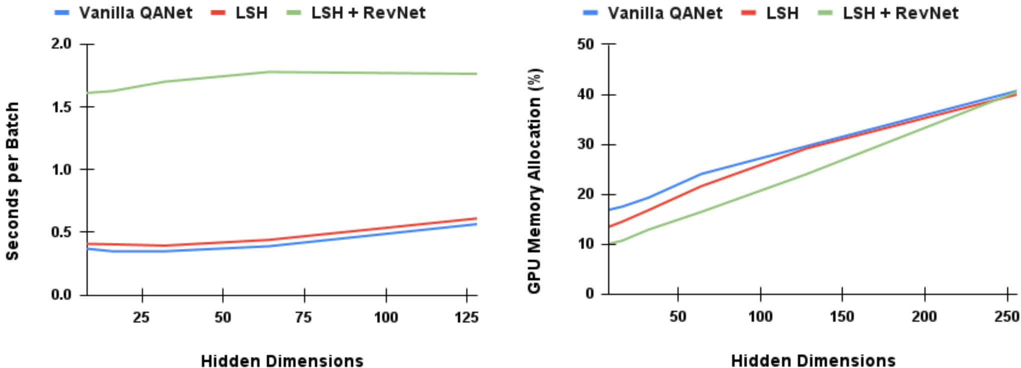


Figure 4: Memory and time consumption of vanilla QANet, QANet modified with LSH self-attention, and QANet modified with both LSH self-attention and reversible transformer modifications (RevTransformer).

In Figure 4, we can see that our modifications were successful in moderately reducing memory consumption, but at a cost to runtime (especially when introducing the RevTransformer blocks).

# 5 Analysis

## 5.1 Error Analysis

We begin by understanding QANet's shortcomings with respect to our F1 and EM performance metrics. A qualitative analysis of our model's performance on the SQuAD 2.0 dev set is summarized in Table 2.

The results are curious. Around 70% of the mistakes made by the both the vanilla and LSH QANet models on the dev set are what we will call "false positives" or "false negatives" (in a roughly

| Error category | Vanilla QANet | LSH QANet |
|---|---|---|
| Total incorrect | 2271 | 2520 |
| False positive | 951 (≈42%) | 1254 (≈50%) |
| False negative | 686 (≈30%) | 555 (≈22%) |
| Bad guess (incl. sub/superstring) | 634 (≈28%) | 711 (≈28%) |
| Substring or superstring | 390 (≈17%) | 431 (≈17%) |

Table 2: Qualitative analysis of Vanilla QANet and LSH QANet errors on the SQuAD 2.0 dev set. The percentages are the approximate percentage of the total error that falls into that category.

even split). In simpler words, when the answer to a particular question is not present in the context paragraph, our model makes a prediction or vice versa (our model will make no prediction despite there being a golden, labeled answer for the question-context pair).

Of the remaining mistakes (what we call "bad guesses"), both models actually comes *close* over 60% of the time. That is, in many "bad guesses", the model predicts some substring or some superstring[2] of the golden label. Inspecting instances of these mistakes, both models often add on or removes articles, adjectives, or punctuation marks from the start and/or end of its prediction relative to the golden label[3]. Some mistakes would be outright acceptable output labels, e.g. *The Russian leader Lenin* instead of *Lenin*. If this behavior were observed in a production system, it would perhaps appear "buggy" to the user, but it likely wouldn't strike them as "incorrect". Note that such mistakes would impact the model's EM score but not its F1 score (quite as much).

This leaves a third and final class of errors (about 30% of the total for both vanilla and LSH QANet) in which the model makes erroneous predictions outright. In some cases, these predictions are at least reasonable (predicting a proper noun where a proper noun is expected, like *Huguenots* instead of *New France*), but some are just plain wrong (e.g. *not completely replace them* instead of *it significantly altered the existing treaties*), though these cases are somewhat rare upon inspection. Another frequent subcategory of error within this umbrella are numerical mistakes, whereby the model predicts an incorrect date or other number. This is interesting as it indicates that the model is sophisticated enough to know a numerical answer is expected, but it lacks the full question understanding necessary to pick the correct one.

### 5.2 Resource Efficiency Analysis

We were pleased to find that reformer-style modifications to QANet improved on memory usage. We were slightly surprised at the scaling we observed, however. Via an ablation study of memory use during a single minibatch (see Figure 4), we saw that memory use didn't seem to *scale* differently when Reformer modifications were applied to the model.

Unfortunately, our LSH+RevTransformer model performed significantly worse in terms of runtime, as seen in Figure 4. Given that the LSH QANet runs only slightly slower than the vanilla QANet, it seems that adding RevTransformer modifications caused the extreme slowdown. The Reversible Transformer modification is supposed to slow runtime somewhat, since backpropagation with RevTransformers requires additional computation to be executed in order to obtain activations (rather than using "cached" activations that are simply stored in memory). However, the level of slowdown we observed is still surprising.

## 6 Conclusion

In this project, we learned that transformer models are powerful tools (relative to traditional, sequence-based models) for question-answering, and can be made to be more resource-friendly via several clever implementational tricks that approximate or obviate the need for the most expensive computations and memory storage.

[2]This is, unfortunately, *not* the first use of the term "superstring" in a non-theoretical physics paper. See e.g. Gorbenko and Popov 2012.

[3]For example, *"Denmark, Iceland and Norway"* instead of *Denmark, Iceland and Norway*, or *like-minded Shia terrorist groups* instead of *Shia terrorist groups*.

Our primary achievement was implementing the QANet transformer architecture from scratch. We subsequently augmented that implementation with Reformer-style component modules to reduce memory use and time complexity of the self-attention computation.

Our most important finding is that applying LSH rather than standard dot-product self-attention in QANet causes a massive performance drop. This suggests that further study of the generalizability of Reformer modifications (and perhaps other transformer efficiency modifications) may need to be studied further and more rigorously. Future work may, for instance, apply Reformer-style modification to popular, expensive language models (like GPT-$X$, BERT, etc.) to make them more accessible and perform a more general study to make sure that these modifications do work for a variety of tasks. Given that these models are often inaccessible to the general populace given their size, the implications of an efficiency-improving transformer modification which preserves performance are immense. Therefore, we believe this to be important work.

## References


[1] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[3] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International Conference on Machine Learning*, pages 4055–4064. PMLR, 2018.

[4] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. Music transformer. *arXiv preprint arXiv:1809.04281*, 2018.

[5] Roshan M Rao, Jason Liu, Robert Verkuil, Joshua Meier, John Canny, Pieter Abbeel, Tom Sercu, and Alexander Rives. Msa transformer. In *International Conference on Machine Learning*, pages 8844–8856. PMLR, 2021.

[6] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *arXiv preprint arXiv:2106.04554*, 2021.

[7] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. Qanet: Combining local convolution with global self-attention for reading comprehension. *CoRR*, abs/1804.09541, 2018.

[8] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[9] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *CoRR*, abs/2001.04451, 2020.

[10] Yi Tay, Dara Bahri, Liu Yang, Donald Metzler, and Da-Cheng Juan. Sparse sinkhorn attention. In *International Conference on Machine Learning*, pages 9438–9447. PMLR, 2020.

[11] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021.

[12] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[13] Sharan Narang, Hyung Won Chung, Yi Tay, William Fedus, Thibault Fevry, Michael Matena, Karishma Malkan, Noah Fiedel, Noam Shazeer, Zhenzhong Lan, et al. Do transformer modifications transfer across implementations and applications? *arXiv preprint arXiv:2102.11972*, 2021.

[14] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603*, 2016.

[15] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[16] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[17] Phil Wang. Lucidrains pytorch reformer. `https://github.com/lucidrains/reformer-pytorch`, 2021.

[18] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Association for Computational Linguistics (ACL)*, 2018.

[19] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.