Beyond DataMUX: Experimentation on LSTM and ALBERT Models

Harbin Hong
Princeton University
harbinh@princeton.edu

Justin Sherman
Princeton University
js119@princeton.edu

Edmund Young
Princeton University
edmund666@princeton.edu

Abstract

In this paper, we explore further applications of data multiplexing (DataMUX), a technique previously introduced in (Murahari et al., 2022) that enables deep neural networks to process multiple inputs simultaneously as a single multiplexed input. Our approach is to use the most successful multiplexing and demultiplexing method from Murahari et al.'s paper, reproduce a portion of the original experiments that used the ROBERTA Transformer base network, and test different base neural networks to judge the generalizability of the DataMUX architecture. In particular, we test the DataMUX architecture on the ALBERT Transformer and an LSTM network using two tasks, sentence classification and named entity recognition. From these experiments, we demonstrated that the current DataMUX architecture is not as effective on the ALBERT model (Lan et al., 2020) and LSTMs. Specifically, we show that ALBERT baseline can multiplex with less than a 10 percent performance drop up to 10 inputs for NER tasks and up to 5 inputs for SST tasks with some performance evaluation drop. For LSTM, we show that it can not multiplex with less than a 10 percent performance drop for NER and only up to 2 inputs for NER tasks. We then discuss the theoretical limitations that may contribute to this drop in performance and discuss possible future steps to be taken in this area of research[1].

1 Introduction

Deep neural networks have been shown to be very capable of modeling a wide variety of natural language tasks. Recent advances in large language models like the introduction of BERT and related architectures demonstrate the potential of such networks with a large number of parameters. However, one major disadvantage of these models is the large computational cost associated with training and processing new inputs. To mitigate this issue, (Murahari et al., 2022) introduce methods to multiplex inputs to these models so that they can simultaneously process and generate predictions for several inputs at a time. The architecture consists of a multiplexer module that combines the inputs, a neural network backbone, and a demultiplexing module to separate the combined output of the backbone into individual outputs. We reproduce the results from the DataMUX paper for two tasks, named entity recognition, or NER, and sentiment analysis, or SST-2. We use the Hadamard transform with a Gaussian vector for multiplexing and the Index Embeddings method for demultiplexing with a RoBERTa transformer backbone network. We selected these particular multiplexing and demultiplexing methods because they were among the most successful in Murahari et al.'s experiments.

In addition to reproducing the trials described above, we sought to evaluate other language models for their capacity to handle data multiplexing. We tested the compatibility of two other neural network backbones with the same multiplexing and demultiplexing methods: an ALBERT Transformer model and an LSTM. By including the latter, we sought to evaluate the extendability of data multiplexing beyond transformer-based architectures.

Using the results from our experiments, we evaluate the performance of DataMUX with backbones of ALBERT and LSTM on the two tasks and attempt to explain possible conclusions from these experiments. Furthermore, our analysis from these experiments suggests multiple potential improvements in this architecture for future research that we believe will lead to many advancements in utilizing deep neural networks efficiently. Hopefully, our analysis could prove to be helpful in identifying key bottlenecks in current DataMUX model architectures that could aid in the creation of more universally applicable and efficient DataMUX models.

[1] https://github.com/hhong00/DataMUXCopy

2 Related Work

Multiplexing is not necessarily a new idea, as it is widely used in signal processing to encode numerous signals through a common transmission. However, until around 2020 when (Lu et al., 2020) developed MUXConv, its uses in deep learning had been fairly small. Even this implementation though was mainly about multiplexing different features and still fed inputs into the model one at a time. (Zhang et al., 2018) proposed a training scheme called mixup where networks were trained with convex combinations of the inputs in order to discover the distributions of labels in the combined representation; the main difference with this implementation though is that the order of the combined instances is not preserved or rediscovered and thus can not be used for inferences. In short, there exists research delving into training schemes and architectures that touch on the idea of multiplexing, but are usually slightly different than actual input multiplexing. Thus, DataMUX was a fairly novel idea that opened up a new area of research for deep learning.

Motivation behind the idea came from the suggestion from studies like (Kaplan et al., 2020), (Allen-Zhu et al., 2020), and more that current deep learning networks are overparameterized and thus could be used more efficiently, in our case for learning representations for multiplexed inputs. (Malach et al., 2020) also were able to show that a fraction of neurons in a trained deep model were usually sufficient which begs the question of whether the remaining portion could be used to capture more complex relations.

Research into the biological side of neural networks in the brain, which many computation deep neural networks take inspiration from, also seemed to show the ability of neurons to be able to multiplex mutliple features as seen in research from (Blumhagen et al., 2011), (Lankarany et al., 2019), and (Rezaei et al., 2023).

Thus, Murahari et al. were the the first to implement this type of input multiplexing and test whether their models could accurately process these mixed inputs for multiple tasks. They experimented with multiple multiplexing methods such as linearly projecting each input using an orthogonal matrix or the Hadamard product with a fixed Gaussian random vector. Also, demultiplexing strategies included using feed forward layers or index embeddings. Although their experiments showed promising results for RoBERTa transformers, their work with image classification with CNN and MLPs on the MNIST classification tasks showed serious performance deterioration as more inputs were multiplexed together.

3 Methods

The multiplexing architecture consists of a multiplexing module, a neural network backbone, and a demultiplexing module.

3.1 Multiplexing module

To combine a batch of inputs $\mathbf{x}^1, \mathbf{x}^2, \dots \mathbf{x}^N$ with each $\mathbf{x}^i \in \mathbb{R}^d$, we apply a Hadamard product with a fixed Gaussian vector $\phi^i : \mathbb{R}^d \mapsto \mathbb{R}^d$ to each $\mathbf{x}^i$ for all $i \in \{1, 2, \dots, N\}$ and take the average to produced the combined multiplexed input. More specifically,

$$\mathbf{x}^{1:N} = \frac{1}{N} \sum_{i=1}^{N} \phi^i(\mathbf{x}^i)$$

where each ϕ^i is the Hadamard product.

3.2 Demultiplexing module

To demultiplex each multiplexed input, we used the index embedding method where we generate index embeddings $\mathbf{p}^i$ which are concatenated to each of the N inputs to be multiplexed before passing into the multiplexing module. Then, after the multiplexed inputs are passed into the neural backbone, the multiplexed hidden logits are passed into an MLP that is trained to separate this multiplexed hidden logit into N indiviudal hidden logits. Each prefix consists of an index token ϵ^i in the i'th position with the remaining tokens being a pad token ϵ^{pad}.

Thus, the prefixes would look something like this:

$$\begin{aligned} prefix^1 &= [\epsilon^1, \epsilon^{pad}, \epsilon^{pad}, \dots, \epsilon^{pad}] \\ prefix^2 &= [\epsilon^{pad}, \epsilon^2, \epsilon^{pad}, \dots, \epsilon^{pad}] \\ &\dots \\ prefix^N &= [\epsilon^{pad}, \epsilon^{pad}, \epsilon^{pad}, \dots, \epsilon^N] \end{aligned}$$

3.3 Neural Backbone

Overall, 3 neural backbone models were used: RoBERTa Transformer model to reproduce the results from the paper, ALBERT Transformer model, and LSTM recurrent neural network model.

3.4 Pretraining Retrieval Warmup Task

Murahari et al. found that adding the multiplexing and demultiplexing layers directly to the model compromised performance. Instead, they rely on a pretraining task to promote the ability of neural backbone models to distinguish the order of multiplexed inputs. In short, the model is first trained to retrieve the tokens in the correct order for a random sequence in the multiplexed input. The training objective for this pretraining retrieval warmup task is given by the following:

$$L_{retrieval}(x^{1:N}) = \sum_{j} -\log \mathbb{P}(w_j^I | h_j^I)$$

All pretraining tasks were done using the wikitext-103 dataset.

3.5 Visualization of Overall Architecture

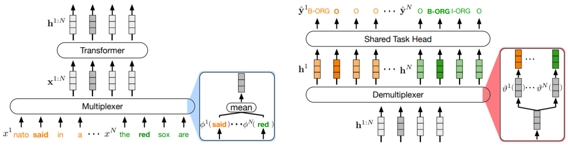
$\mathbf{h}^{1:N}$
Transformer
$\mathbf{x}^{1:N}$
Multiplexer
x^1 nato said in a ⋯ x^N the red sox are
mean
ϕ^1(said)⋯ϕ^N(red)
$\hat{\mathbf{y}}^1$ B-ORG O O O ⋯ $\hat{\mathbf{y}}^N$ O B-ORG I-ORG O
Shared Task Head
$\mathbf{h}^1$ ⋯ $\mathbf{h}^N$
Demultiplexer
$\mathbf{h}^{1:N}$
ϑ^1( )⋯ ϑ^N( )

Figure 1: Illustration showing overall architecture of DataMUX model with Transformer backbone on NER task. Performs multiplexing on inputs using Mutliplexer module before passing into backbone Transformer. Multiplexed hidden logits passed into Demultiplexer module to yield individual hidden logits and then passed into Shared Task Head to yield predicted entities.

Figure 1 shows an example of a Transformer DataMUX model working on an NER task. Inputs are passed into the Multiplexer module on the bottom left where they become multiplexed representations of the inputs and then passed into the neural backbone, in this case a Transformer. Hidden multiplexed logits are received from the Transformer which are then passed into the Demultiplexer module to be separated into their individual hidden logits before being passed into a shared task head that extracts the predicted tag from the hidden logit.

4 Experimental Setup

4.1 Models

We first used the same ROBERTA Transformer model for the neural backbone that Murahari et al. used for their Transformer based DataMUX model with the same hyperparameters. This transformer consisted of a a 12-layer model with hidden dimension size of 768 and 12 self-attention heads. After reproducing these results, we used an ALBERT Transformer model with the same number of layers, hidden dimension size, and self-attention heads. Finally, we switched the backbone to a bidirectional LSTM with the same number of layers but hidden size being half of 768 to account for the bidirectionality. We compare the performance of each of these DataMUX models to the baseline performances of the DataMUX models without any multiplexing. These baselines were produced by doing first doing the pretraining retrieval warmup task on each DataMUX model with the number of inputs multiplexed equal to 1 ($N = 1$). This is similar to the second baseline, $B2$, done in Murahari et al.'s original paper.

4.2 Tasks

We evaluated our models and baselines on a subset of the same token-level classification and sentence-level classification tasks used in Murahari et al.'s experiments. We used token-level classification on the CoNLL-2003 Named Entity Recognition (NER) task Sang and Muelder (2003) and sentence-level classification through sentiment classification SST-2 Socher et al. (2013).

All DataMUX models are pretrained using the retrieval warmup task on the Wikitext-103 dataset (Merity et al., 2016) and this retrieval loss training objective is continued to be used as another objective during the finetuning tasks with the total loss being a weighted sum of finetuning task loss and retrieval loss.

$$\mathcal{L} = 0.1 * \mathcal{L}_{retrieval} + 0.9 * \mathcal{L}_{task}$$

5 Implementation

In this section, we will go over the technical details to implement each of the three DataMUX models tested: RoBERTa backbone DataMUX model, ALBERT backbone DataMUX model, and LSTM backbone DataMUX model.

5.1 RoBERTa DataMUX

Thanks to the codebase being uploaded on GitHub with clear instructions on the readme file, we were able to start reproducing the experiments early in our project timeline. One primary concern though was the number of steps that the training scripts used. If we had left the number of training steps constant at 500,000, each pretraining task for each N value, where N represents the number of inputs to be multiplexed, would have taken 16 hours on a V100 GPU from Google Colab Pro's premium GPU. Since there are three backbones and each backbone has 5 N values to test, this would have meant that to complete the pretrain tasks alone for every backbone and N value, we would have needed 240 hours worth of compute units, which is obviously infeasable since this would roughly require $312 dollars for just this part alone. Instead, after communicating with Vishvak Murahari, we reduced the number of steps to 50,000 after confirming that retrieval loss should still be suitably low for up to $N = 10$. Thanks to this reduction, each pretraining task time was reduced to around 1 hour and 30 minutes, and the finetuning tasks for SST-2 took around 2 hours for each N value while the finetuning NER tasks took the longest at 6 hours each.

Another problem came from Google Colab's idle timeout error, but a workaround that we found was to run a simple for loop to print a message every 10 seconds for however long we needed to run the training and finetuning tasks.

5.2 ALBERT DataMUX

This DataMUX model was relatively easy to implement. After changing the RoBERTa config file to a new ALBERT config file with the correct hyperparameters and then loading an ALBERT model from the Huggingface Hub instead of the RoBERTa model, we were able to run the same scripts with little difficulty to obtain results for pretraining and finetuning tasks across all N values. These experiments took similar amounts of time as the RoBERTa DataMUX models.

5.3 LSTM DataMUX

This ablation was the most troublesome to implement. The first step was to switch out the backbone for an LSTM with approximately similar hyperparameters and modify the corresponding sections of the forward method. The specific hyperparameters of the model consisted of the following: embedding dimension of an intermediate embedding layer was 768, the number of LSTM layers was 12, the LSTM was bidirectional and had a hidden size of 384 to have a concatenated hidden size of 768 with the bidirectionality. Then, we had to find an alternative to the checkpoint creation functionality for Transformers that the DataMUX architecture uses for starting the finetuning task from the checkpoint after the pretraining retrieval warmup task. In the end, we simply hardcoded boolean variables that dictated whether the LSTM implementation would be used and if so, would save the model weights and parameters to a checkpoint file created using the 'torch.save' command and could later be loaded with the 'torch.load' command. We found that making a zip file of the model parameters and saving that zip file to a Google Drive folder was very useful to get around Google Colab's runtime deletion. Afterwards, we were able to use the same scripts as in the original codebase to run the LSTM implementation. Note that although these scripts still load hyperparameters from the 'Roberta.config' file for the RoBERTa model, they are never actually used since we also hardcoded the LSTM hyperparameters into the code.

5.4 Additional Implementation Details

During the finetuning tasks for $N = 5$ for all 3 DataMUX architectures, we also saved the average retrieval and task loss of every 100 steps to create graphs of both retrieval and task loss as the number of steps increases.

6 Experiments

6.1 F1 Score/Accuracy

As seen in Figure 2, we were able to successfully reproduce the RoBERTa DataMUX experiments for up to $N = 20$ in both NER and SST-2 tasks. Compared to the baseline, even with only 50,000 pretrain and finetune train steps, the accuracy did not drop a significant amount for SST-2; for NER, f1 scores actually increased, which as mentioned by Murahari et al., points to the robustness of this model from possible regularization due to mixup[2]. In contrast, ALBERT and LSTM models were not able to perform to the same extent. ALBERT DataMUX models performed on a similar trend to RoBERTa DataMUX models. For instance, in the NER task, ALBERT DatMUX model was able to actually outperform the baseline for small values of N up to 5 and performed similarly to the baseline, within 0.05 f1 score, for $N = 10$. For SST however, ALBERT could only perform similarly up to $N = 5$, within 0.1 accuracy score, before

[2] Zhang et al

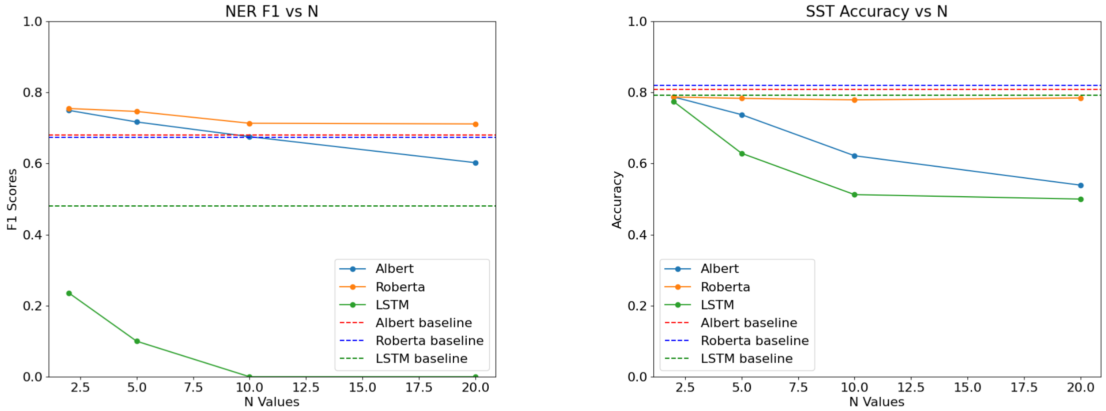

Figure 2: Left figure shows NER F1 scores across all 3 architectures for $N = 2, 5, 10, 20$ compared to baselines of the various architectures which appear as dotted lines according to the legend. Right figure shows SST-2 accuracies across all 3 architectures for $N = 2, 5, 10, 20$ compared to baseline of the various architectures which appear as dotted lines according to the legend.

performance grew markedly worse, getting close to random[3] at $N = 20$.

LSTM DataMUX model unfortunately was significantly worse than its baseline for both tasks and every N value except for $N = 2$ on the SST task. In fact, NER F1 scores for $N \geq 10$ dropped to 0, indicating that the classifier is so bad that some tags are never being generated. One probably reason for this striking deficiency is that NER places significantly more emphasis on retrieval loss than SST-2 does because of the nature of the tasks. Thus, as seen in Figure 4, this retrieval loss is much more detrimental in these tasks which most likely leads to the performance of the LSTM DataMUX model in NER and SST-2 tasks.

6.2 Retrieval and Task Loss Relation

In order to further analyze the results of the f1 scores and accuracy, we created graphs of the retrieval loss and task losses as a function of the number of training steps during the $N = 5$ SST-2 task as shown in Figure 3.

From this analysis, we found an interesting trend in the ALBERT DataMUX architecture. It seems that until retrieval loss decreases to a suitably low number, task loss would not decrease as seen around the 3,000 step mark where ALBERT task loss was relatively constant until the corresponding retrieval loss decreased to around 1. This observation intuitively makes sense because until the multiplex and demultiplex modules can truly start separating accurately, training on the task will not yield any insightful results. However, something that is perhaps more surprising is that there seems to be a critical value that the retrieval loss has to reach beforehand. We were unable to confirm the existence of this critical value with the other architectures because RoBERTa DataMUX model pretraining brings the retrieval loss well below and LSTM DataMUX model pretraining never comes close to the apparent critical value from the ALBERT DataMUX model.

6.3 Pretraining Loss

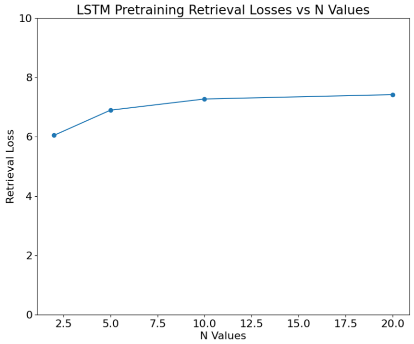

Figure 4: Figure showing Retrieval Loss from Pretrain as N Values Increase

Finally, we show that this result of being unable to multiplex and demultiplex accurately for LSTMs was not limited to just this $N = 5$ value by plotting the retrieval loss during the pretraining retrieval

[3] Accuracy ≈ 0.5

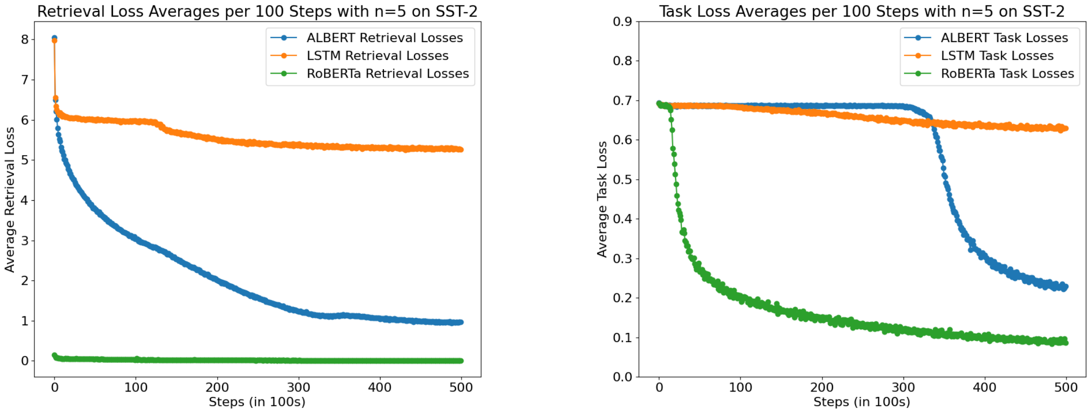

Figure 3: Left figure shows average retrieval loss per 100 steps across all 3 architectures throughout the 50,000 steps during finetuning experiment for $N = 5$ on sentence classification SST-2 task. Right figure shows average task loss per 100 steps across all 3 architectures throughout the 50,000 steps during finetuning experiment for $N = 5$ on token classification NER task.

warmup task for all N, as seen in Figure 4. This graph seems to confirm again the intuition linking this retrieval loss and task loss.

7 Results

7.1 Number of Epochs

From Figure 2, we saw that up to $N = 10$, the ALBERT DataMUX model was able to perform relatively similarly to its baseline for the NER task and run through 862.07 epochs of its evaluation dataset compared to only 114.16 for the baseline. Because LSTM did not perform close to the baseline for NER for all values of N, there is little point in including the number of epochs that these multiplexed models ran through compared to the baseline 114.16.

For the SST-2 task, the ALBERT DataMUX model performed similarly to the baseline up to $N = 5$, at which it ran through 119.05 epochs of the evaluation dataset, compared to only 10 epochs for the baseline. The LSTM DataMUX model performed similarly up to $N = 2$ at which it ran through 47.53 epochs compared to 10 for the baseline.

Of course, greater N values still lead to greater number of epochs of the evaluation dataset which also means greater throughput, but bearing in mind the cost to f1 score or accuracy, these results are not of interest nor valuable to us.

7.2 Comparison to Original Paper

Although we may not have been able to reach the same results as the RoBERTa Transformer DataMUX models for ALBERT Transformer DataMUX models, just the fact that little to no variation was needed to multiplex up to 5/10 inputs for SST-2/NER tasks is an interesting result, especially considering how fewer parameters ALBERT has compared to RoBERTa. We believe these results suggest that an increase in the number of training steps in both the pretraining and finetuning tasks could potentially help to better fit the data. Perhaps then if ALBERT is found to perform at more similar levels after increasing the number of training steps, one could be left with the choice of whether they need to prioritize the number of parameters or running time.

Unfortunately, LSTM DataMUX model was not able to perform well for most N values. From looking at the Figure 4, we can surmise that the multiplexing and demultiplexing strategies are unsuited for this LSTM DataMUX model. Upon reflection, this makes sense as well because an LSTM naturally loses information after each layer unlike Transformers which uses its self-attention heads to extract a hidden logit representation over all states. This loss of information in the LSTM could be causing a partial loss of the multiplexed information on the index embeddings from the demultiplexing strategy we used which would of course make separating the multiplexed hidden logit into individual hidden logits much more difficult. In which case, future research into making this LSTM DataMUX model work would have to find a way of embedding positional information into the inputs while making sure that the LSTM layers do not accidentally cause loss of this information.

Task	N	LSTM Accuracy/F1 Score	ALBERT Accuracy/F1 Score	RoBERTa Accuracy/F1 Score
SST-2	1	0.7928	0.8085	0.82
	2	0.774	0.7873	0.787
	5	0.6288	0.7375	0.7833
	10	0.5125	0.6219	0.7792
	20	0.4875	0.5391	0.7844
NER	1	0.4806	0.6801	0.6732
	2	0.2356	0.7495	0.7548
	5	0.1178	0.71695	0.7465
	10	0	0.6756	0.7133
	20	0	0.6024	0.7114

Table 1: Numerical accuracies or F1 scores respectively for SST-2 and NER depending on which N value is used and which backbone model is used for the DataMUX experiments.

7.3 Limitations

Some limitations of this paper include the number of experiments we did along with the number of training steps we used.

Because of the limit on both computation costs as well as monetary costs[4], we limited ourselves to only SST-2 and NER tasks. It is very possible that the ALBERT DataMUX model could perform better in some of the other tasks from the original DataMUX paper, such as MNLI, QQP, or QNLI. This would be an easily implemented future test that could possibly yield interesting results. Specifically, inference tasks such as MNLI or QNLI could yield different results than what we have seen already.

In addition, the number of steps taken were drastically reduced in our experiments, but this also resulted in less training for the pretraining retrieval warmup task and the finetuning tasks. For RoBERTa, this did not pose much of a problem as the pretraining retrieval loss usually decreased to a very small amount as seen in Figure 3. However, in models like ALBERT and maybe even LSTM, the extra 450,000 steps might have made a big difference in comparing the difference between the performance of RoBERTa DataMUX models vs ALBERT DataMUX and LSTM DataMUX models. For RoBERTa DataMUX models, the training loss would have decreased asymptotically to 0 at probably a faster rate. However, at the end of the 500,000 steps, ALBERT may have also decreased asymptotically to 0 or to enough of a degree that there may not have been much of a difference in retrieval and task loss between the two DataMUX models. In this case, assuming one has a dedicated GPU he or she could use, ALBERT DataMUX with 500,000 steps would almost always be a better alternative compared to RoBERTa DataMUX model given an almost identical performance with less parameters and thus a decreased time to train all the parameters. In addition, while I do not foresee LSTM DataMUX models becoming on par with the two Transformer DataMUX models just from increasing the number of steps to 500,000 because of the reasons I listed before about the difference in retaining information of the hidden states in LSTMs, an increase in performance is still plausible given that in Figure 3, we see a slight decrease in retrieval loss for the LSTMs as the steps go by.

8 Conclusion

In this paper, we recorded the results of reproducing the RoBERTa DataMUX model to the best of our abilities given our computing environment from Murahari et al.'s "DataMUX: Data Multiplexing for Neural Networks" and our analysis of the two modifications we made. Our two modifications were to change the neural network backbone from a RoBERTa Transformer to an ALBERT Transformer and then to an LSTM. From the two modifications, we found results indicating that in terms of direct transferability, ALBERT DatMUX models were most similar, but our data suggested that LSTM DataMUX models are possible with better multiplexing and demultiplexing strategies.

Future work can test more specialized multiplexing and demultiplexing strategies, experiment with different baseline neural networks, and also test to see if there truly is a critical value that retrieval loss must decrease to before task loss starts decreasing.

[4] For reference, the amount of experiments we did cost $250 using Google Colab Pro

9 Acknowledgements

This is a final project for COS484 Spring 2023. We thank Professor Danqi Chen for the semester and her work in teaching us this topic. We also thank all the graduate and undergraduate teaching assistants for their help during office hours in guidance and implementation details. Finally, we thank Vishvak Murahari for his communications with us throughout the project to better help with our understanding of the experiments made.

References

Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. 2020. Learning and generalization in overparameterized neural networks, going beyond two layers.

Francisca Blumhagen, Peixin Zhu, Jennifer Shum, Yan-Ping Zhang Schärer, Emre Yaksi, Karl Deisseroth, and Rainer W Friedrich. 2011. Neuronal filtering of multiplexed odour representations. *Nature*, 479:493–498.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations.

Milad Lankarany, Dhekra Al-Basha, Stéphanie Ratté, and Steven A. Prescott. 2019. Differentially synchronized spiking enables multiplexed neural coding. *Proceedings of the National Academy of Sciences*, 116(20):10097–10102.

Zhichao Lu, Kalyanmoy Deb, and Vishnu Naresh Boddeti. 2020. Muxconv: Information multiplexing in convolutional neural networks.

Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. 2020. Proving the lottery ticket hypothesis: Pruning is all you need.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models.

Vishvak Murahari, Carlos E. Jimenez, Runzhe Yang, and Karthik Narasimhan. 2022. Datamux: Data multiplexing for neural networks.

Mohammad R. Rezaei, Reza Saadati Fard, Milos R. Popovic, Steven A. Prescott, and Milad Lankarany. 2023. Synchrony-division neural multiplexing: An encoding model. *Entropy*, 25(4):589.

Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. mixup: Beyond empirical risk minimization.