CS 7643 Project Report : Universal Transformer

Nicholas Beshouri, David Decker, Seungyoon Lee, Kara Bethany Liu
Georgia Institute of Technology
{nbeshouri3, ddecker30, slee3271, kliu360}@gatech.edu

Abstract

The Transformer in "Attention is All You Need" (Vaswani et al.*, 2017) [5] aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease, and this novel architecture outperforms previous RNNs based designs and is widely used in NLP. However, Transformers struggle with some more algorithmic language tasks where the recursive bias of RNNs seems to be helpful. The authors of "Universal Transformers" (Dehghani et al., 2019) [2] try to address these limitations, and here we implement their architecture in PyTorch and attempt to replicate their results of better performance of Universal Transformers on several tasks as described in the paper. We successfully replicated results on some tasks, but not all, and present our experience, results, and analysis.*

1. Introduction/Background/Motivation

Until recently, RNNs were the default choice for sequence tasks like language modeling and machine translation. But in the last few years, they've largely been replaced by parallel, feed forward architectures like Transformers that achieve higher performance on many sequence tasks and can be trained more quickly than RNNs, which must execute time steps sequentially. However, Transformers struggle with some more algorithmic language tasks where the recursive bias of RNNs seems to be helpful, and they have difficulty dealing with sequence lengths not encountered during training. The authors of "Universal Transformers" (Dehghani et al., 2019) [2] try to address these limitations with a generalization of the original architecture combining self-attention blocks with a repeated transition function that is shared across input positions and time steps and claim to achieve state of the art results on a variety of language tasks. In our project, we implemented their architecture in PyTorch and attempted to partially reproduce their results on four different natural language understanding tasks. Reproducing results is important for validating published research findings.

The first task was the bAbI question answering task, which requires a model to achieve near perfect accuracy on a synthetic dataset that tests 20 different aspects of language understanding thought to be prerequisites for conversing fluently with humans (Weston et al., 2015) [6]. The second is the copy, reverse, and addition algorithmic tasks ("Neural GPUs Learn Algorithms" Kaiser and Sutskever, 2016) [3]. This takes strings composed of decimal symbols ('0'-'9') as an input and performs copy, reverse and integer addition to output the results as a string, where the data was generated as described in the "Neural GPUs Learn Algorithms". The paper presented three tasks, but we added a binary addition task as well to demonstrate the strength. The third is the LAMBADA task ("The LAMBADA dataset", Paperno et al., 2016) [4]. This requires the agent to read passages from novels of up to 202 words and choose the correct last word from nearly 200,000 possibilities. The missing last words are designed to be hard if a person were to only read the last sentence but easy if they have read the entire passage. We also planned to do a translation task, where passages of English are to be translated into German, but after some false starts we decided to focus on the other three tasks.

2. Approach

To replicate the paper's results, we first implemented a Universal Transformer (UT) class in PyTorch. We wanted to keep the interface as similar as possible to PyTorch's native nn.Transformer class to enable abstraction in our training script. So, we had our UT class inherit from that class, and only added arguments as necessary, like maximum steps and model halting threshold. A wrapper class was also built to add details like embeddings for inputs and outputs around the execution of the transformers. Next, a working draft of the bAbI task training script was developed and shared with team members. The other team members then leveraged this code as needed for their own tasks. We each then iterated on running their code on colab, fixing bugs, and hyperparameter tuning as time allowed, before summarizing the results to present.

While we didn't use other repositories directly, we did reference `https://github.com/tensorflow/tensor2tensor` and https://github.com/andreamad8/Universal-Transformer-Pytorch for inspiration and sanity checking.

In the following subsections, we describe the training and evaluation approach to each task.

2.1. Architecture

The UT proposed by the paper is similar to the original transformer described by Vaswani et al. (2017) (Vaswani *et al.*, 2017) [5] except that its weights are tied across its layers so that the same transformation is applied iteratively to each input vector as it moves through the network. The number of iterations can either be fixed or vary dynamically between positions using a dynamic halting mechanism that calculates a "halting probability" at each timestep and, after this probability crosses a set threshold, begins to simply copy the input vector from the previous iteration without updating it further.

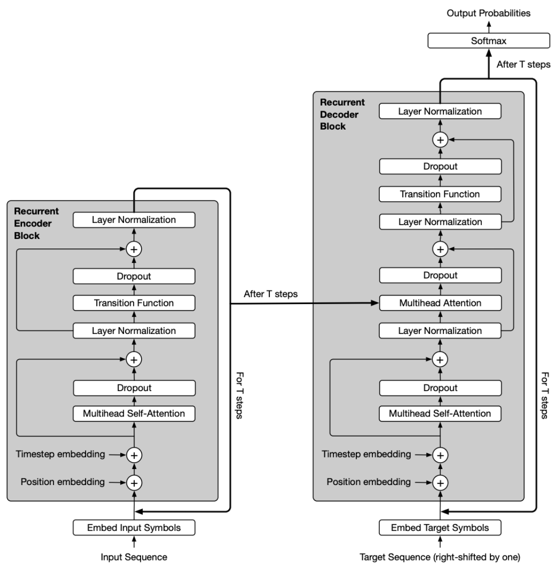

Figure 1. Universal Transformer architecture

The UT also adds a depth embedding to encode a vector's depth in the network, which it applies along with the positional embedding before the self attention layer at each iteration, and a configurable "transition function" similar to the fully-connected layer at the top of a standard transformer layer. This can either be a pair of fully-connected layers with a RELU between them or a pair of depthwise separable convolutions (Chollet, 2016) [1], also separated by a RELU.

We were able to implement the UT in PyTorch by passing a custom encoder and decoder to nn.Transformer. Instead of looping through a set of separate transformer layers with their own parameters like nn.TranformerEncoder and nn.TranformerDecoder, our encoder and decoder apply one stack of operations repeatedly. The operations themselves are similar to what's applied in nn.TranformerEncoderLayer and nn.TransformerDecoderLayer, with minor modifications to accommodate the transition function and extra embedding.

To implement the dynamic halting mechanism, we translated the tensorflow code on page 14 of the original paper to PyTorch. To make the halting loop differentiable, the process is cast as a weighted sum where the nth term is the result of applying the multi-head attention and transition function to the input n times. The weight applied to each term is chosen by a fully-connected layer based on the previous state and constrained so that once the cumulative weight reaches the halting threshold, the weight for all subsequent steps is 0.

In the following subsections, we describe the training and evaluation approach to each task.

2.2. bAbI Question and Answer Task

The bAbI dataset consists of twenty different tasks, each of which has a one-thousand and a ten-thousand example variant. The tasks can also be trained all together or individually, with the one-thousand example, multi-task version being the most difficult. In the original paper, the authors evaluated all 4 conditions (one-thousand multi-task, ten-thousand multi-task, one-thousand individual, ten-thousand individual) and presented average test set error rates on 10 different runes. Given our limited compute, we focused on two different conditions: the cross-trained one-thousand per task condition and the ten-thousand example of the particularly difficult "3 Supporting Facts" task, which we believed particularly suited to the UT and dynamic halting. In both cases we made a best effort to tune comparably good hyper-parameters on both the UT and Vanilla Transformer (VT). After tuning, we collected test set results for 5 different runs after training the model for 400 epochs with early stopping.

2.3. bAbI Question and Answer Task 3 (WhereWasObject) Tuning

We decided to focus and tune one individual b4bI task – the most to understand how UTs would perform over VTs. In this task, the model aims to predict where an object was (past tense) given 3 supporting facts amidst various noisy facts. More details are in Appendix A. We trained these models specifically on the 10K dataset for this task. The paper highlighted that UTs outperformed VTs because of three reasons: 1. attention distributions get sharer in later steps when the model picks up on relevant supporting facts, 2. dynamic halting allows the model to vary the number of processing steps or ponder time by the number of supporting facts required and 3. ponder times at different positions is less uniform for tasks requiring more supporting facts and containing longer stories, indicating that the model is capable of ignoring irrelevant facts.

2.4. Algorithmic Task

In Dehghani et al., 2019, the authors pointed out that VT performs poorly on Algorithmic tasks compared to LSTM, and showed that UT outperforms LSTM and VT by a wide margin. To replicate this, three learners were implemented and compared for each task - copy, reverse and addition (integer and binary). The training and test data was generated by following the method described in ”Neural GPUs Learn Algorithms”. For the accuracy measure, the final test set with length of 400 were validated by character by character accuracy and sequence by sequence accuracy. They used sequences of length 40 but we also tried sequences of length 8 to see if the sequence length affects the performances.

2.5. LAMBADA Task

In Dehghani et al., 2019, the LAMBADA task was approached in two ways: once with a standard language model, and once with a reading comprehension model which selected from only previously used words in the passage, which provided better accuracy. An examples can be seen in B We attempted the standard language model approach. As such, during training, the model outputs logits across its entire vocabulary for each subsequent word in the passage after the first one, and gets feedback for back propagation from the true label and cross entropy loss. In contrast, for the validation and test sets we only track and report the accuracy and perplexity of the last word of the passage, since this is the focus of the task. The repeated transition function may help the UT generalize from the language model to the more challenging task without fine tuning.

2.6. Challenges

One broad challenge was that each task's setup was unique and required careful reading and interpretation of both UTs and VTs for the task from Vaswani et al., 2017 and Dehghani et al., 2019. Some details were left out of Dehghani et al., 2019, like which transition function they used for a particular task, and no hyperparameters were shared. The LAMBADA training data set was just a collection of novel passages, some of which were thousands of words long. This would be too large for training, so the original instances of the text were split into lengths of no more than 203 words at time, to match the longest length of a task passage. Other details like attention and padding masks and loss functions also had to be adjusted or developed for each task.

Another challenge for some tasks was runtimes. We dealt with this by running very small subsets of the data first to verify code, although that did not catch all problems, such as a data leakage issue in the LAMBADA task until larger datasets were used. The LAMBADA task also required runtimes of about 10 hours per epoch with the full dataset. To fit in more epochs per run, we sampled 500,000, or just under half of the original 203 word samples described above.

Task id	VT(ours)	UT(ours)	UT(original)
1	0.6%	1.9%	0.10%
2	54.2%	70.0%	4.30%
3	70.7%	79.0%	14.3%
4	8.4%	23.3%	0.4%
5	16.3%	20.2%	4.3%
6	3.0%	6.3%	0.8%
7	21.4%	24.5%	4.1%
8	14.0%	17.5%	3.9%
9	1.3%	6.9%	0.3%
10	11.4%	20.9%	1.3%
11	9.3%	22.6%	0.3%
12	3.8%	7.0%	0.3%
13	6.4%	7.2%	1.1%
14	40.7%	60.4%	4.7%
15	0.0%	7.9%	10.3%
16	54.8%	55.2%	34.1%
17	42.0%	40.7%	51.1%
18	5.3%	6.1%	12.8%
19	91.2%	91.3%	73.1%
20	0.0%	0.3%	2.6%
Average	**22.7%**	**28.5%**	**11.2%**
Tasks solved	**7**	**3.4**	**14**

Table 1. Comparison of Results Across Algorithms and Boards

A third challenge was the limitations imposed by Google Colab. Even with Colab Pro subscriptions, we were limited to a maximum of three running notebooks at a time. The GPUs also had memory limitations within them, which limited our batch sizes. This limited one degree of freedom of our hyperparameter tuning and prevented using larger batch sizes for faster runtimes on some of the larger tasks. There is also a 24 hour run limit, which prevented us from doing more than just a few epochs of LAMBADA, even with a smaller training dataset.

On hyperparameter tuning, model performance varied greatly across runs. This was also highlighted in Appendix E of the paper's results. The variance across runs was large, and differed across different bAbI tasks for example. As a result, tuning over average runs was important and this required a lot more time.

3. Experiments and Results

3.1. bAbI Question and Answer Task

Unfortunately, our UT failed to outperform the VT on the multi-task, 1K examples version of the bAbI dataset. Its average total error rate over 5 runs was 28.5%, 5.8 points

Model	Validation Data	Test Data
VT	46.3%	46.1%
UT	59.1%	59.3%
UT w/ Dynamic Halting	31.8%	29.2%

Table 2. bAbI average error rate over all runs (3 for this work, 10 for the original paper), 10K examples

higher than the VT and 17.3 points higher than the UT in the original paper. The results on the individual tasks were similar except for task 17 (positional reasoning), where our UT beat the vanilla by 1.3 points and the author's UT by 10.4 points.

The cause of these differences is unclear. It's likely that the original authors were able to explore more of the hyperparameter space than we were, and it's possible that getting good performance from the UT requires more careful tuning, though our VT consistently solved 7 tasks, 1 more than the author's, which suggests they didn't tune that exhaustively (or if they did, they did so unfairly, though admittedly that's another issue).

Another possibility is that there's a flaw in our UT implementation. While some aspects of the UT like the early halting mechanism required complex custom code, the core of the idea was relatively simple to implement using pre-built PyTorch layers and even without halting should have solved 12 bAbI tasks, a feat we were unable to replicate. That said, it seems likely there were at least some subtle differences in our implementation and that these differences, perhaps combined with our relatively coarse tuning, were behind the performance gap.

3.2. bAbI Question and Answer Task 3 (WhereWasObject) Tuning

For task 3, the UT with Dynamic Halting performed the best as seen in Table 2, similar to in the paper. However, regular UTs without dynamic halting failed to outperform VTs. This is likely attributed to insufficient tuning - VTs are simpler to tune compared to our implementation of UTs, which have a much greater number of hyperparameters and take longer lead times to train. UTs are capable of very complex modeling. However, there is a tendency to overfit as observed by the validation loss being much higher than that of the train loss, and the challenge lies in preventing this, especially over smaller sets of data.

The key hyperparameters in Table 3 that affected UT model performance were the learning rate and batch size. Small learning rates and batch sizes had to be used to address overfitting for UTs. For VTs, given their less complex nature, a larger batch size and learning rate could be used. This made training time much longer, however. A high number of epochs not only would increase training time but also contribute to overfitting as data would be passed through the model multiple times. Therefore, 150 epochs

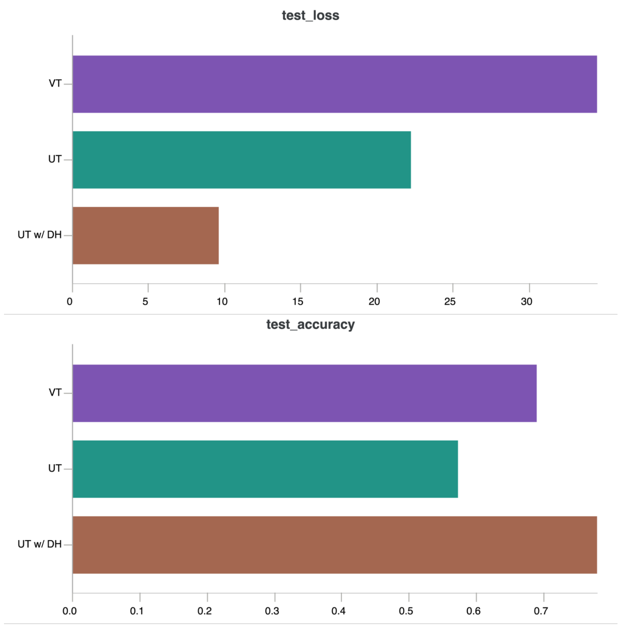

Figure 2. Loss and accuracy results from the test set for bAbi task 3

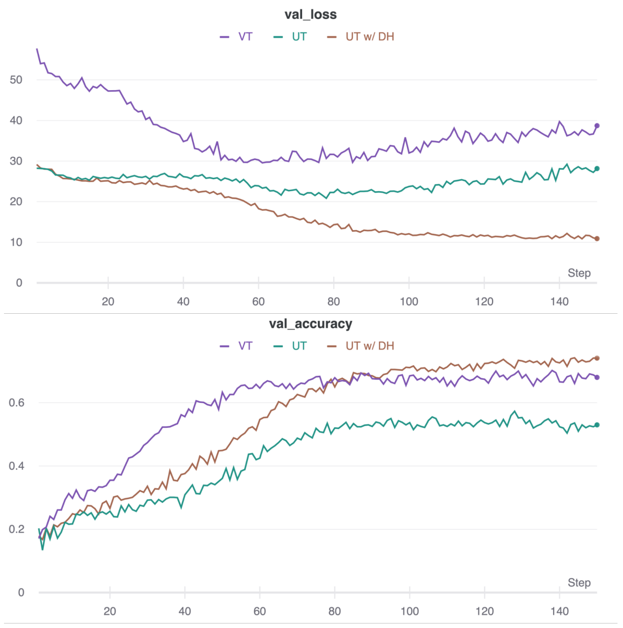

Figure 3. Loss and accuracy results from the validation set for bAbI task 3

was selected to be compared across all models. It is noted that across all models, Adam outperformed RMS Prop as the optimal optimizer. As for the number of attention heads, selecting 2 or 3 heads was optimal across all models. Therefore, 2 heads were selected as it introduced less complexity.

The optimal number of maximum steps selected for dynamic halting was 6. Too many would introduce additional complexity and increase train times while too little would

Hyperparameters	VT	UT	UT-Dyn. Halt.
Epochs	150	150	150
Batch Size	64	32	32
Learning Rate	0.00005	0.0001	0.00005
Optimizer	Adam	Adam	Adam
Model Halt Threshold	-	-	0.9
Max Steps	-	2	6
Dyn. Halt. Loss Weight	-	0.001	0.001
Number of Heads	2	2	2
Layers	1	-	-

Table 3. Optimal Hyperparameters for bAbI task 3

reduce the model's performance. For a UT without dynamic halting, a lower number of maximum steps was therefore optimal.

3.3. Algorithmic Task

We set up the end-to-end data generation, training, validation and accuracy measurement architecture for LSTM, VT and UT and compared the training, validation, and test loss for each task per learner. We validated our architectures on sequence of length 8 for a sanity check and tried the actual sequences of length 40 as described on the paper and measured the loss and accuracy when it was used for the sequence of length 400. To be fair and to account the constrained resource, we limited all of the learners to 10 epochs. Vaswani et al., 2017 used Adam for the optimizer but it didn't give us better results than SGD in this limited setup, hence we used SGD instead for Transformers.

Below are the loss and accuracy plots on each task. We wanted to prove that UT does better than other models while VT does worse than LSTM. We approached this on how the cross entropy loss changes per training and also how the final model performs on the test set by sequence and character level. All the raw data from the experiments were logged over wandb and are available here. We tried two transition types for UTs - fully connected type "fc" and depth wise conv type "dwc".

3.3.1 Copy task

Our sequence accuracy result is not the same as in Dehghani et al., 2019, but character accuracy agrees well. As seen in Figure 4, the validation loss indicates very strong performance on UT with depth wise convolution transition, and its character level accuracy outperforms others by a wide margin.

3.3.2 Reverse task

In the same hyperparameter setting as the other tasks, our LSTM does poorly on reverse task while the Transformers does well. We also find that UTs do better than VTs. As noted in the challenge above, it was not possible to tell

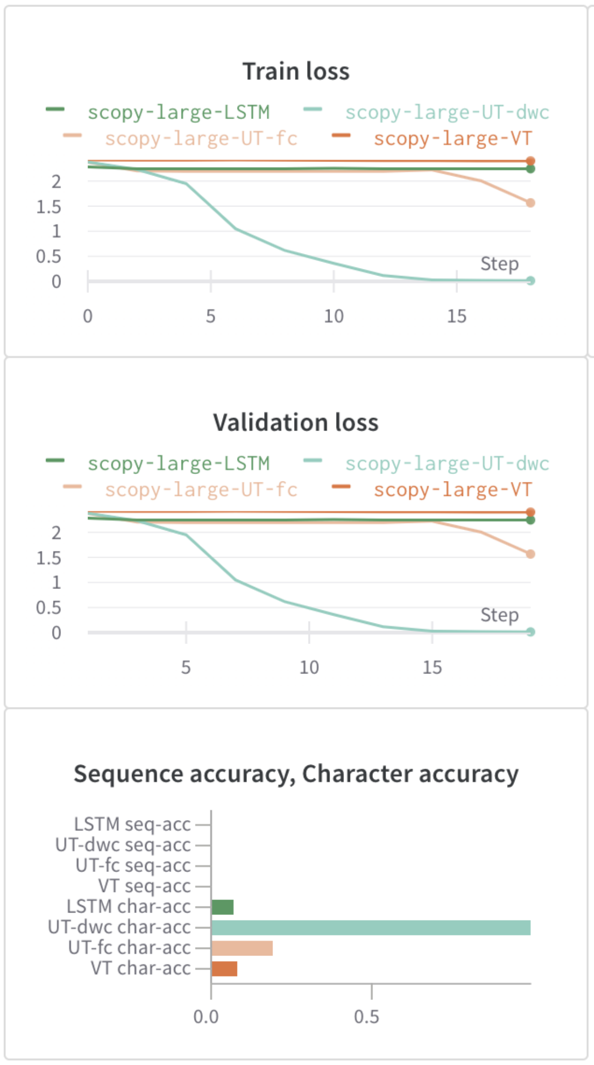

Figure 4. Loss and accuracy results from copy task

what type of LSTM architecture was used although "Neural GPUs Learn Algorithms" hinted LSTM+A was used as described in "Grammar as a Foreign Language" for another task. As shown in Figure 5, we are able to see that UTs converge much quicker than others. When they were tested on the same length of sequence (length = 40), a UT with depth wise conv transition shows 100% character level accuracy while LSTM only shows 11.8%, and VT shows 10.1%.

3.3.3 Addition (integer) task

Addition task presents the strength of UTs. Fully connected transition type UTs exhibit 97.8% character level accuracy and outperforms LSTMs and VTs by a wide margin. Both UTs converge pretty quickly. As seen in Figure 6, when the trained model was used for the same length of sequence (length = 40), UT shows 100% accuracy while

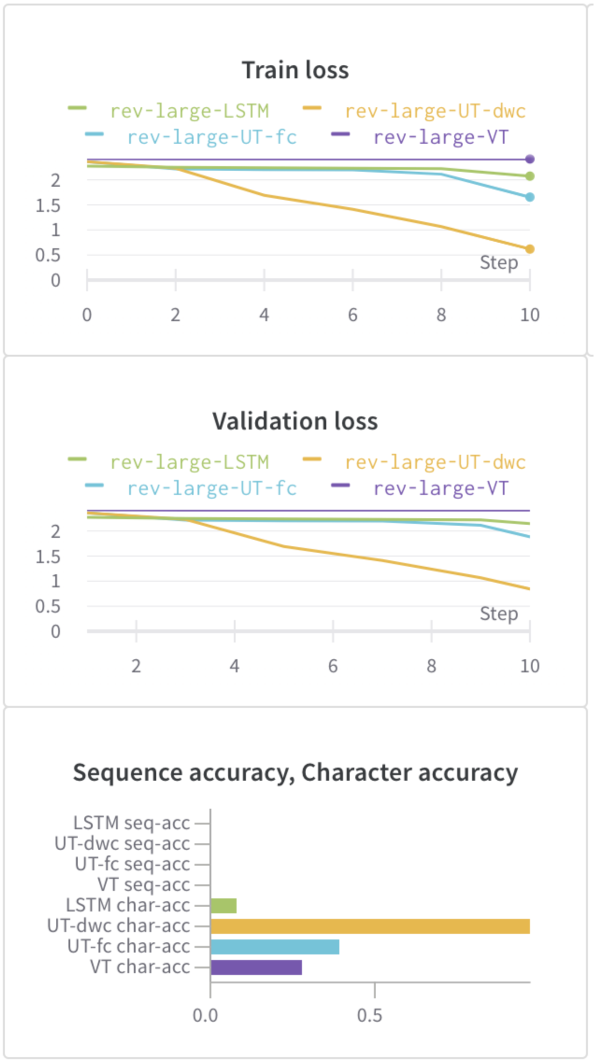

Figure 5. Loss and accuracy results from reverse task

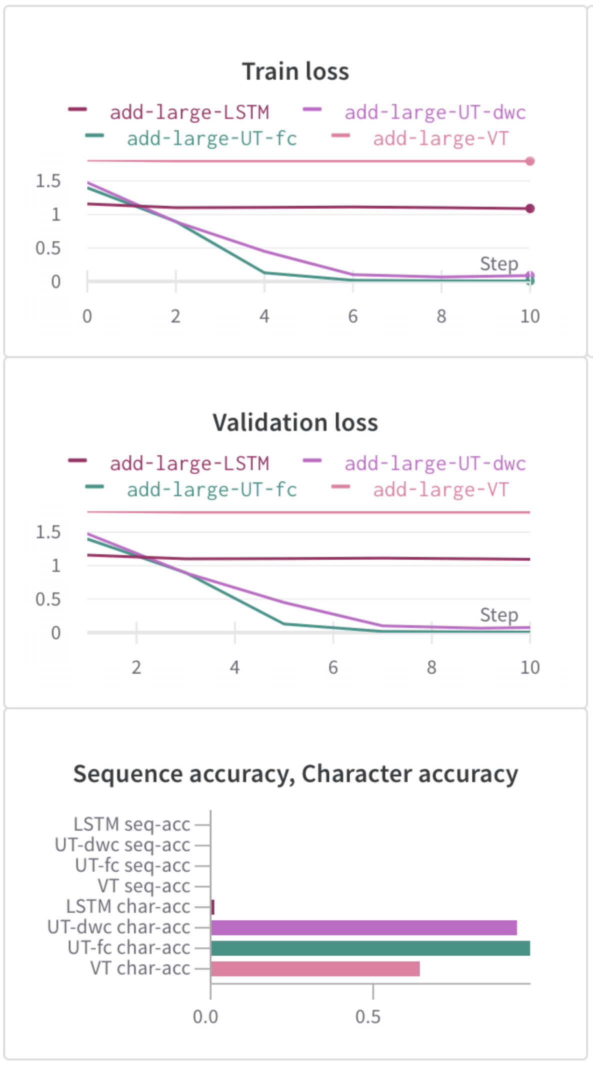

Figure 6. Loss and accuracy results from integer addition task

LSTM shows 56.3% and VT shows 56.1% character level accuracy.

3.3.4 Addition (binary) task

Although the authors did not show the performance comparison on the binary addition task, we made a comparison - it also shows the strength of UT with 99.9% and 99.3% character level accuracy as in Figure 7. This is yet another good task to demonstrate the strength of UT. It runs strong without overfitting and also generalizes well on unseen data which has different lengths of sequence.

3.4. LAMBADA Task

This proved to be a difficult task to complete, given the training data and sequence sizes. After debugging, we finally only had time for one run each for the UT and the VT, for just four epochs each, which is all we could fit within the 24 hour colab limit. Our run time for UT was over 18 hours and was over 23 hours for VT.

For the experiments, we trained a UT and a VT as standard language models to just predict the next word in each passage in the training data. To do this, we passed in the same sequence as the source and the target, with masks that hid subsequent words. The data came tokenized, and a file with word frequencies was available for the dataset to build our vocabulary, which was used to index the tokens. The models learned embeddings during training and both used the Adam Optimizer. We used batch sizes of 24 because that was the largest we could safely fit in the V100 GPU. A learning rate of 0.0001 and dropout of 0.1 were chosen based on previous experience. Both models used model dimensions of 512, again for space considerations. We used 8 attention heads because it is a common number seen in

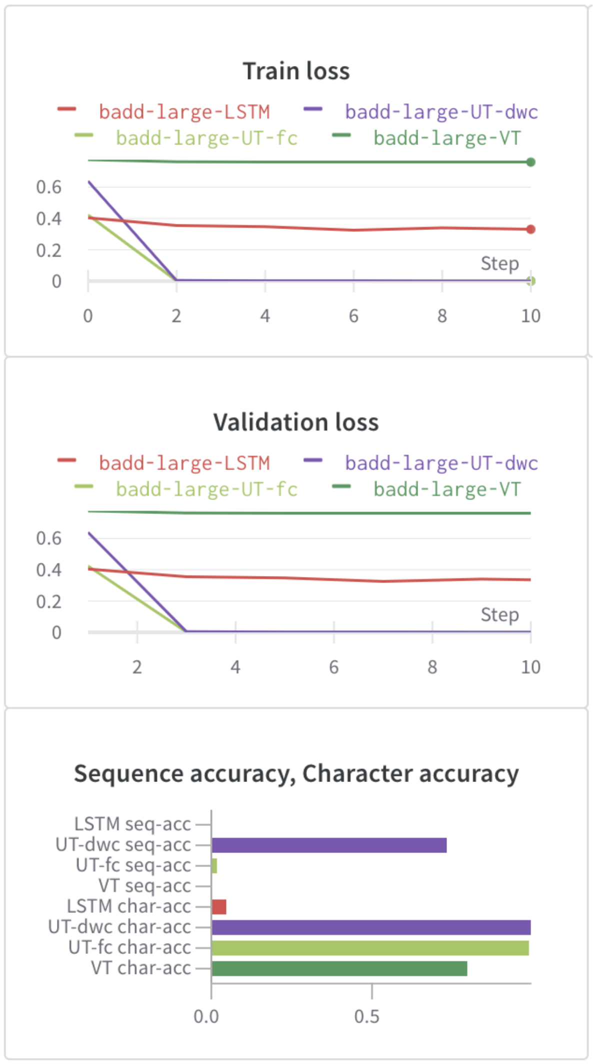

Figure 7. Loss and accuracy results from binary addition task

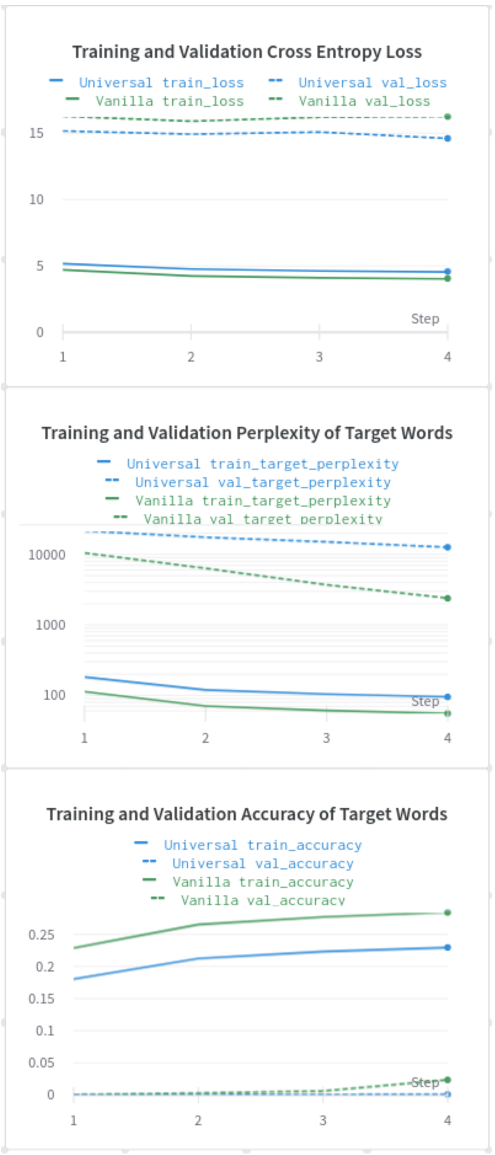

Figure 8. Loss and accuracy results from the LAMBADA task

NLP models, and the maximum steps for the UT was chosen to match the best results from Dehghani et al., 2019. We applied the fully connected transition function for this task.

In the top panel of Figure 8, we can see the per-word cross entropy loss for the training and validation sets for each model averaged across all words in the passages. The middle panel shows the perplexity of just the target words, which were the last word of each passage. Because the training set was composed of arbitrary passages of 203 words, the target words were no more challenging than the average word. The final panel displays the training and validation accuracy of the target words.

There are a couple of things to note from this limited experiment. First, there's clearly a lot of variance between the training and validation results for both models, so this suggests that we would consider lowering the learning rate and increasing regularization. If we were to continue, we could experiment with limited epochs to see which hyperparameters appeared to provide a good bias-variance tradeoff before expanding to longer runs. Also, despite that, the models are clearly improving on the LAMBADA task over the epochs.

With such limited experiments, we cannot draw firm conclusions about the UT on this task, but we present our final results in Table 4. These perplexity scores are one to two orders of magnitude larger than the results presented in Dehghani et al., 2019. Also, the better performance of the VT compared to the UT without tuning may provide more evidence that UTs are more sensitive to hyperparameter tuning.

	dev		**test**	
	ours	original	ours	original
VT	2,389 (0.023)	5,122 (0.0)	11,179 (0.0)	7,321 (0.0)
UT	12,612 (0.001)	279 (0.18)	22,753 (0.0)	319 (0.17)

Table 4. Perplexity and accuracy (in parentheses) comparison between VT and UTs on the LAMBADA dev and test sets. The UTs use 6 steps.

4. Conclusion

We were able to demonstrate the strength of the UT on Algorithmic tasks and on bAbI task 3. Specifically for task 3, we found that UTs, especially those using dynamic halting, performed the best. However, we were not able to demonstrate that the UT does better than VT on the bAbI question and answer multi-task and LAMBADA task given the limited resources. Although we did not replicate the UT's performance in all tasks in Dehghani et al., 2019, we feel confident that we understand UT and its general training pipeline for these tasks well, which we consider a success.

If we had more time and compute resources, we would do more hyperparameter tuning and more training epochs. We would also like to apply UTs to other tasks like machine translation. For the LAMBADA task, which had much longer sequences, we would like to look at methods such as Extended Transformer Construction[**?**].

5. Work Division

The delegation of work among team members are in Table 5.

A. bAbI WhereWasObject (Task 3)

. . .
20 John dropped the apple.
21 John grabbed the apple.
22 John went to the office.
26 John travelled to the office.
25 John journeyed to the bathroom.
27 Sandra left the milk.
28 Mary went to the bedroom.
. . .
38 John discarded the apple there.
39 Where was the apple before the bathroom? office 38 25 22

B. LAMBADA example

Context:
```
"Yes, I thought I was going to lose the
baby."
"I was scared too," he stated,
sincerity flooding his eyes.
"You were?" "Yes, of course.  Why do
you even ask?"
"This baby wasn't exactly planned for."
```
Target sentence:
```
"Do you honestly think that I would
want you to have a _______?"
```
Target word:
```
miscarriage
```

C. code and data repositories

https://github.com/dfd/CS7643_group_project/
https://github.com/nbeshouri/universal_transformer
https://github.com/seungyoon/CS7643_group_project/
https://github.com/Karawkz/universal_transformer

References

[1] François Chollet. Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357, 2016. 2

[2] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. *CoRR*, abs/1807.03819, 2018. 1

[3] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015. 1

[4] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *CoRR*, abs/1606.06031, 2016. 1

[5] 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. 1, 2

[6] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015. 1

Student Name	**Contributed Aspects**	**Details**
Nicholas Beshouri	Implemented the Universal Transformer Class	Wrote the PyTorch code for our implementation for the Universal Transformer class.
	Executed the bAbI Task	Wrote code from scratch to train and evaluate universal and vanilla transformers on the bAbI dataset.
	Paper Writing	Wrote the introduction, bAbI results, and architecture sections.
David Decker	Executed the LAMBADA Task	Leveraged the bAbI task code and wrote code for the Lambada data preparation, training, and evaluation.
	Paper Writing	Put together the initial outline/draft of the paper; wrote sections related to the LAMBADA task; LaTeX formatting.
Seungyoon Lee	Executed the Algorithmic tasks	Wrote code from scratch to prepare data, train and evaluate LSTM, universal and vanilla transformers on Algorithmic tasks.
	Paper Writing	Wrote sections related to the Algorithmic tasks, Abstract and Conclusion; LaTeX formatting.
Kara Bethany Liu	Hyperparameter tuning for the bAbI task 3 What Object Modeling	Tuned and compared best-performing models to match paper's results.
	Attempted the Machine Translation Task	Initial code to clean and vectorise dataset.
	Paper Writing	Initial architecture section; wrote sections related to task 3 hyperparameter tuning.

Table 5. Summary of Team Member Contributions