Our Attempts in the LMSYS Chatbot Arena Human Preference and Building a Neural Network From Scratch

Alekhya Katragadda(alekhya@bu.edu) and Yawen Zhang (cywz@bu.edu)

Abstract

In this paper, we walk you through our processes of training models that will predict human preferences in the LMSYS Chatbot Arena Human Preference. For the competition, we focused on training three different models inspired by the codes that were made available on the LMSYS Chatbot Arena Competition forum.
We first started by testing the Gemma-Keras-Inference notebook and the DeBERTa Keras NLP notebook. Both our attempts at fine-tuning these pre-trained models were time-consuming and we were not able to obtain the best results we had hoped for. We then decided to quit using any pretrained models that were available and instead decided to build our own Neural Network, a small one but efficient. It was when using our own model and training it that produced better results.
We attempt to show you that building relatively small models (ours was a mere 5 layers) can give you results close to the ones that are produced when using large pre-trained models. While our at-home model gives us a very humble result of 1.098, it is rather impressive that we can get comparable results.

Introduction

The rapid development of large language models (LLMs) is transforming the industry and how we interact with technology. To optimize LLMs effectively, it is important to align these models with human preferences for meaningful usage. This report presents our participation in the LMSYS - Chatbot Arena Human Preference Predictions competition, which aims to bridge the gap between LLM capabilities and user expectations by predicting which model response users will prefer.
Our approach began with an extensive literature review to familiarize ourselves with strategies from existing research on reinforcement learning from human feedback and preference modeling. We then developed and tested multiple notebooks to enhance model performance. One of our initial attempts, despite significant efforts, encountered out-of-memory errors. However, through iterative improvements, we eventually produced a successful notebook that overcame these challenges and yielded promising results.

Literature review:

Introduction

https://www.kaggle.com/code/emiz6413/inference-gemma-2-9b-4-bit-qlora

This notebook serves as an instruction for the competition, effectively evaluating and comparing two models' responses to text prompts. The author uses the 4-bit quantized Gemma-2 9b instruct model with a LoRA adapter. This review will explain the notebook's methodology, emphasizing its data processing techniques, configuration, and inference techniques.

Configuration

In the notebook, several critical libraries and tools are installed to facilitate the model inference process: **'transformers'**, **'peft'**, and **'accelerate'**.
Transformers: the library from Hugging Face provides pre-trained models and tokenizers essential for handling NLP tasks.

Peft: represents for parameter-efficient Fine-Tuning, adapting models with fewer parameters for specific tasks.
Accelerate: simplifiers distributed computing and accelerates training and inference processes.
This notebook operates on two parallel GPUs. This supports efficient processing and large-scale computations, crucial for handling the intensive workloads.

Data processing
Data preprocessing involves applying a custom **'process_text'** function to clean and format text. The preprocessing ensures that text is in a suitable format for subsequent tokenization and model inference.
This step is important for maintaining data quality, which directly impacts model performance.
Tokenization and model loading
Text is tokenized using the **'GemmaTokenizerFast'** without applying test-time augmentation (TTA). This process converts the text into input that the model can understand.
.

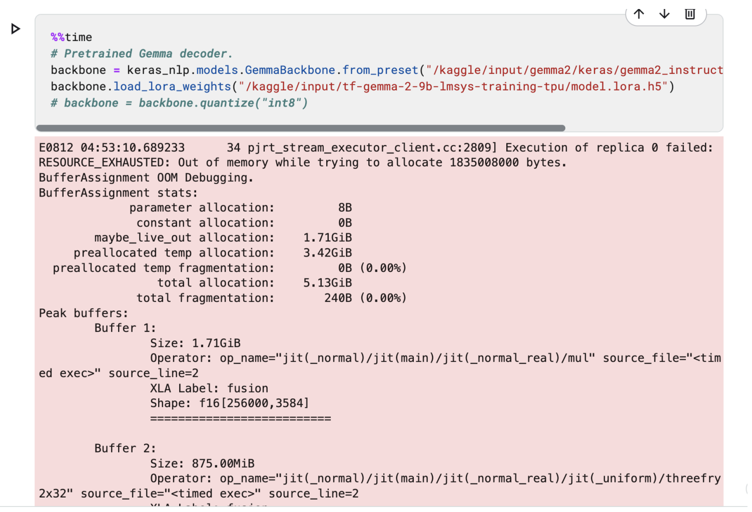
Our Attempt at Various Models:

1. Gemma-Keras-Inference notebook

The Gemma-2 model is loaded on two separate GPUs. The model is adapted using a LoRA adapter to enhance its performance.

Inference process
Inference is performed in batches, with results accumulated and processed to generate predictions. Parallel processing is applied to handle large datasets efficiently. This notebook sets turn off the TTA, which reduces computational overhead and model robustness.

Summary
The notebook reports a log loss of 0.9371 on the validation set and a public leaderboard score of 0.941. The inference process takes approximately 4 hours with a maximum sequence length of 2048 tokens.
By integrating the Gemma-2 model with a LoRA adapter and employing techniques like parallel processing and TTA, the notebook presents a well-rounded solution handling large-scale inference tasks. It inspired our work on optimizing processing time and using TTA to improve model performance
This notebook uses gemma2 model architecture with tools like TensorFlow and Keras to handle large-scale text data. We add the explanation to the code file. We encountered an out-of-memory


```
%%time
# Pretrained Gemma decoder.
backbone = keras_nlp.models.GemmaBackbone.from_preset("/kaggle/input/gemma2/keras/gemma2_instruct
backbone.load_lora_weights("/kaggle/input/tf-gemma-2-9b-lmsys-training-tpu/model.lora.h5")
# backbone = backbone.quantize("int8")
```

```
E0812 04:53:10.689233      34 pjrt_stream_executor_client.cc:2809] Execution of replica 0 failed:
RESOURCE_EXHAUSTED: Out of memory while trying to allocate 1835008000 bytes.
BufferAssignment OOM Debugging.
BufferAssignment stats:
             parameter allocation:         8B
              constant allocation:         0B
        maybe_live_out allocation:    1.71GiB
     preallocated temp allocation:    3.42GiB
  preallocated temp fragmentation:         0B (0.00%)
                 total allocation:    5.13GiB
              total fragmentation:       240B (0.00%)
Peak buffers:
        Buffer 1:
                Size: 1.71GiB
                Operator: op_name="jit(_normal)/jit(main)/jit(_normal_real)/mul" source_file="<tim
ed exec>" source_line=2
                XLA Label: fusion
                Shape: f16[256000,3584]
                ==========================

        Buffer 2:
                Size: 875.00MiB
                Operator: op_name="jit(_normal)/jit(main)/jit(_normal_real)/jit(_uniform)/threefry
2x32" source_file="<timed exec>" source_line=2
```

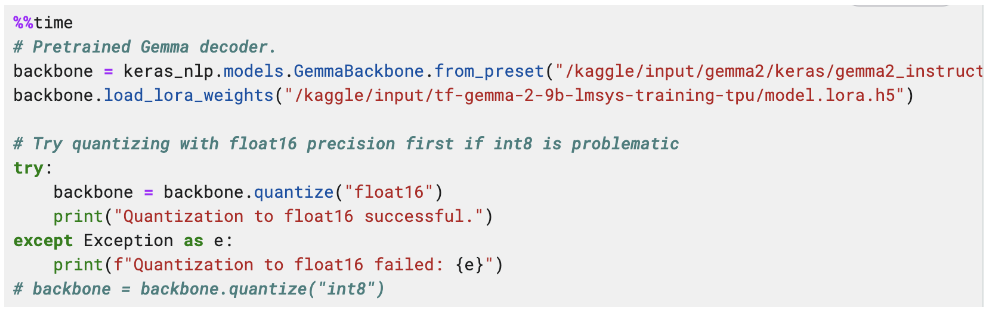
issue when we tried to load the model. Below are the steps we took to address the problem.

Initial attempts:
Check system memory: first, we ensured that the system had sufficient memory to handle the model and its operations. After confirming the memory availability, the issue persisted.
Reduce batch size: we lowered the batch size from 8 to 4, and further to 1. Smaller batch sizes require less memory, but this change did not resolve the issue.
Lower sequence length: We reduced the sequence length from 1024 to 512 to decrease memory usage, but the problem continued.
Advanced techniques:
Model quantization: We attempted to reduce memory usage through model quantization. Initially, we used ‘ **int8**’ quantization with the following code:

```
backbone = backbone.quantize("int8")
```

This method converts model weights to ‘**int8**’ precision, which typically reduces memory footprints. However, the conversion likely caused large intermediate allocations, and the issue remained unresolved.
‘**Float16**’ quantization:

As an alternative, we tried ‘**float16**’ quantization to reduce memory usage while avoiding the problems encountered with ‘**int8**’. Unfortunately, this did not solve the problem either.
Reduce memory fraction:
To prevent the model from using all available memory, we reduced the memory fraction from 1.0 to 0.5, limiting memory usage. Despite this adjustment, the out of memory error persisted.
Additional measure:
We made sure to clear unused variables and perform garbage collection using ‘**gc.collect()**’ to free up memory. However, these efforts did not fix the issue.



```
%%time
# Pretrained Gemma decoder.
backbone = keras_nlp.models.GemmaBackbone.from_preset("/kaggle/input/gemma2/keras/gemma2_instruct
backbone.load_lora_weights("/kaggle/input/tf-gemma-2-9b-lmsys-training-tpu/model.lora.h5")

# Try quantizing with float16 precision first if int8 is problematic
try:
    backbone = backbone.quantize("float16")
    print("Quantization to float16 successful.")
except Exception as e:
    print(f"Quantization to float16 failed: {e}")
# backbone = backbone.quantize("int8")
```

2. Neural Network Built From Scratch

We were inspired to start working on a simple neural network after coming across this code where the author also built a simple neural network from scratch. https://www.kaggle.com/code/bhanupratapbiswas/lmsys

Preprocessing the data:

Combining the prompt with the response helps us understand the relationship between the input and the output of the system. This is essential for the chatbot for text classification. This also ensures the model learns with context and not random stand-alone text.

In the code we use text_a and text_b textual features to combine the prompts which will help the model understand the input and the output with context. Then we used the TF-IDF (Term Frequency Inverse Document Frequency) method to convert the text into numerical features. After which the data is combined into a feature matrix, a target y is created because this is a multi-class classification setup where we need to predict one of three outcomes (winner a, b or tie).

The data is then split into training and validation with 20% of the data for validation. We do this to prevent overfitting.

To improve upon this preprocessing method: it would be better if the feature extraction and combination were using more advanced techniques like embeddings or sequence models so that they would be able to capture the relationships of the order of works or the meanings more intricately.

We also use the StandardScaler function to scale the data to unit variance. This ensures that all features contribute equally to the model in order to prevent larger scaled from dominating the learning process. There are also other techniques available that could be used in the place of StandardScaling such as Min-Max scaling and Robust scaling. I believe using better methods in feature extraction and vectorization would have helped us tune our model better.

Training the model:

While training our model, we apply a grid search to tune the neural network model. We focus on various dropout rates, activation functions and the density of neurons in each layer.

Grid Search:

In our grid search, we use three key hyperparameters (drop out rates, density values and activation functions) to evaluate the model's performance. The dropout rates are a regularization technique that is supposed to help with overfitting, the density values tell us about the number of neurons in the layers and this directly affects the model's capacity to learn. We also use various activation function, which we shall discuss now.

Activation Functions:

The activation functions in neural networks are very important in order to understand the hidden nodes to produce a more desirable output. The main purpose of an activation function is to introduce non linearity into the model. There are two different types of activation functions: Linear Activation Functions and Non-linear Activation Functions.

The non-linear activation functions are the most widely used activation functions. They make it easy for the model to generate or adapt with a variety of data and to differentiate between he outputs.

One of the most popular activation functions is the sigmoid function, which we also use here.

Sigmoid:

The sigmoid activation function exists between 0 and 1. It is used for models that need to predict probabilities. But the sigmoid also has something called the *Vanishing Gradient Problem* which tells us that the more the layers are added to the neural network the gradients of the loss function approaches zero and makes it difficult to train.

The sigmoid function, for instance, tries to normalize all the various large and small input values between zero and one. Therefore this large change in the input will cause the derivative to become small. And gives us a small output.

But this is not always a problem! For small neural networks, this should not be a problem.

The neural network we designed has 5 layers, making it quite shallow which we hope is not influenced by the vanishing gradient problem.

The simplest problem for sigmoid's vanishing gradient problem is the ReLU which we shall discuss next.

ReLU:

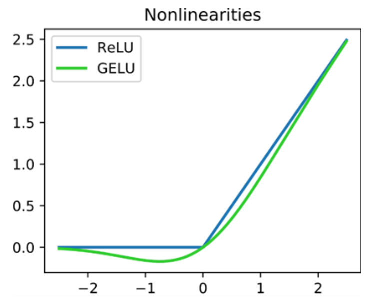


image from Wikipedia: https://en.wikipedia.org/wiki/Rectifier_(neural_networks)#/media/File:ReLU_and_GELU.svg

Rectified Linear Unit, ReLU activation function is also known as a ramp function. It is a piecewise linear function that outputs the input if it is positive and zero if it is negative.

Unlike sigmoid, it doesn't suffer with Vanishing Gradient Problem. But it does have the draw back if "dying ReLU" where the neurons become inactive and stop learning if they consistently output zero.

Tanh:

This is another activation function that we are applying in our grid search. As the name status, it is a hyperbolic tangent function. Similar to the sigmoid, it outputs vary from -1 to +1. It is usually used in the hidden layers of the neural networks.

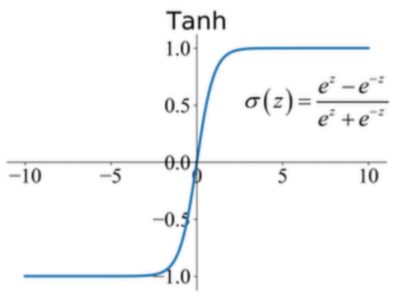


Image from: https://paperswithcode.com/method/tanh-activation

It is preferred over the sigmoid function because it gives a better performance for multi-layer neural networks. But this also unfortunately suffers from the same Vanishing Gradient Problem, and as we saw above that ReLU doesn't.

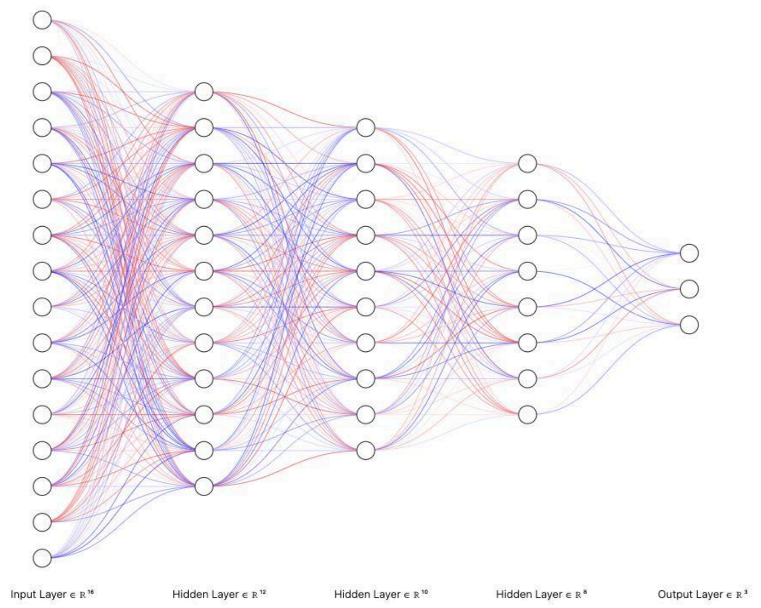
Softmax:

We use the softmax activation function in the last layer of our pyramidal neural network. The softmax function takes in a vector of raw outputs of the neural network and returns a vector of probability scores.

It returns an output vector which has the same number of entries as the number of classes.The reason we use softmax for our last layer is because we need to return 3 classes (Winner_a, Winner_b and Winner_tie). Therefore using softmax as the last activation layer makes more sense.

$$f_i(\vec{a}) = \frac{e^{a_i}}{\sum_k e^{a_k}}$$

The equation denotes that softmax is a ratio of the standard exponential input vector function



Original image generated from: https://alexlenail.me/NN-SVG/

and the sum standard exponential output vector function over all the classes in a multi-class classifier.

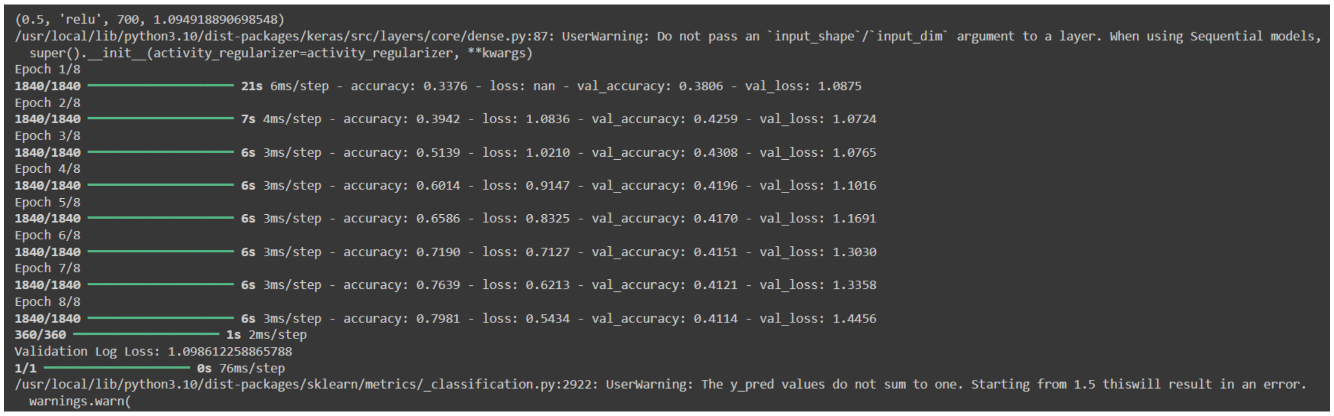
Tanh is a smoother function compared to ReLU, which means it changes gradually with input. But it can saturate for very high or very low which makes the gradients smaller. But it is more effective than sigmoid function because it is zero-centered in the hidden layers. ReLU does not saturate in the positive direction which helps with not facing vanishing gradient.

The above figure represents (non-scalable) the pyramidal structure of the neural network that was programmed. We traverse through the layers until we reach the last softmax layers which caters to the 3 class labels that we have to determine.

By including all these activation functions in the grid search we compare and contrast how these functions impact learning of the model. ReLU leads to quicker learning due to it's fast convergence and non saturating nature. But Tanh is more zero centred and smooth giving us a more stable training.

After doing the grid search we always ended up with these hyperparameters:

1. Best Dropout=0.5 (or sometimes 0.7)
2. Best Activation Function= ReLU
3. Best Density= 700 (or sometimes 500)
4. Loss= 1.094



```
(0.5, 'relu', 700, 1.094918890698548)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models,
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 21s 6ms/step - accuracy: 0.3376 - loss: nan - val_accuracy: 0.3806 - val_loss: 1.0875
Epoch 2/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.3942 - loss: 1.0836 - val_accuracy: 0.4259 - val_loss: 1.0724
Epoch 3/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.5139 - loss: 1.0210 - val_accuracy: 0.4308 - val_loss: 1.0765
Epoch 4/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6014 - loss: 0.9147 - val_accuracy: 0.4196 - val_loss: 1.1016
Epoch 5/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6586 - loss: 0.8325 - val_accuracy: 0.4170 - val_loss: 1.1691
Epoch 6/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7190 - loss: 0.7127 - val_accuracy: 0.4151 - val_loss: 1.3030
Epoch 7/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7639 - loss: 0.6213 - val_accuracy: 0.4121 - val_loss: 1.3358
Epoch 8/8
1840/1840 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7981 - loss: 0.5434 - val_accuracy: 0.4114 - val_loss: 1.4456
360/360 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Validation Log Loss: 1.098612258865788
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:2922: UserWarning: The y_pred values do not sum to one. Starting from 1.5 thiswill result in an error.
  warnings.warn(
```

Training:

While training the model we do multiple iterations over the entire dataset. We do 8 epochs and used a batch size of 25. We found that due to time, memory and processor constraints, 25 batch size gave us fair results. During the training the model minimizes over the loss function so that it can make accurate predictions.

Validation:

For the at home model we split the data into 80-20. 80% of the data (of 57000) was used for training the model and 20% was used for validation. This also like training is passed to the fit method. The predictions by the model are compared to the true labels to calculate the loss and accuracy. We tried our best to monitor the validation in order to prevent overfitting. Post training, the model makes predictions on the validation set using the predict method. The results are stored in a results list along with the corresponding hyperparameters.

Lastly, we find the best configuration based on the best loss value. Then using this we build the final neural network model, the layers are similar to the grid search but with the best found parameters. The output layer for this model is also a softmax making it suitable for multiclass classification.

When compiling the model, we use the Adam optimizer, categorical_crossentrop for the loss function and we track the accuracy. The model as mentioned is trained over 8 epochs with a batch size of 25. Based on this we finally make the best prediction.

3. DeBERTa Notebook

https://www.kaggle.com/code/awsaf49/lmsys-kerasnlp-starter

With the DeBERTa Keras NLP starter code that was pinned by LMSYS, we tried to change small parameters to observe the difference.

In this model we added K-fold cross validation, Gradient Accumulation, changed the ratio at which we split the data fro testing and validation. We made a 60-40 split. With 60% for testing anf 40% for validation. We had to do this split.

How we could've improved this model:

1. We could've increased the model depth and added more layers. This could have helped the model learn complex patterns in the data.
2. We also could have done Batch Normalization which could've helped us stabilise and speed up training the input layers. This also would've helped us converge faster.
3. We also could've used other optimizers other than Adam, like SGC or RMSprop. Also increasing the number of epochs would've helped us with overfitting by implementing early stopping.
4. Implementing better preprocessing methods would have helped us train the model better. We think adding noise to the data by paraphrasing the data would have helped.
5. We also could use better regularization techniques. This would have helped us penalise large layers and help prevent overfitting.

Results:

Comparing DeBERTa and Neural Network Built From Scratch

The five-layer simple neural network offers a very good contrast when compared to the results of the DeBERTa extra small encoding model (which has 12 layers). The DeBERTA model performs significantly worse with a log loss of 3.34 compared to the simple neural networks 1.0986. We speculate this is due to two main reasons:

DeBERTa's significantly increased complexity (with twice as many layers) without any regularization could lead to a heavily overfitted model—this would explain the extremely poor performance on test data. In the simple model we present we specifically tune the dropout rate and layer density through a grid search—which in turn serves to address overfitting to an extent. This helps us find the right combination of parameters which allows us to maximize performance while reducing the risk of overfitting. It is, however, important to note that our grid search is only limited to a 5 x 4 x 3 (density x dropout rate x activation function) array due to computational resource limitations. We could possibly get better results for a larger grid search, where not only are these parameters varied, but the network depth is varied as well.

On the other hand, The DeBERTa model used is a pre-trained out of the box model—it used standard out of the box settings, such as a Gaussian Linear Unit activation function and an Adam optimizer. Given that it was an out of the box model, it is reasonable to expect somewhat poor performance on training tasks compared to a moderately fine-tuned five layer network—our second model, where we searched for different densities, dropout rates and activation functions. If this were a more long term project, it would have been feasible to fine tune different parameters of DeBERTA such as its optimizer or activation function. Allowing for better performance.

Why we think the Neural Network Built from scratch did better:

We use a very simple validation process. For the DeBERTa model we use a 60/40 (train/test) split and for the simple neural network we use an 80/20 (train/test) split. The split for DeBERTa was lower because we were trying to find a combination that would allow the model to run in a reasonable amount of time, so we went with a smaller training set. The five-layer network was simple enough, so we were able to use an 80/20 split with more training data. The validation accuracy is simply calculated as a proportion of predictions the model gets right in the validation set.

For the number of questions solved, we use the following formula:

number of total samples x proportion of validation set (giving us the number of samples in the validation set) x validation accuracy in the final run of the model

Plugging the numbers in for the DeBERTa extra small model, we get: 57,000 x 0.4 x 0.3076 = 7013.28 (rounded up to 7014 questions solved)

Similarly, for the simple five-layer network, we get: 57,000 x 0.2 x 0.7981 = 9098.34 (rounded up to 9099 questions solved)

References:

https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

https://builtin.com/machine-learning/relu-activation-function https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

https://www.pinecone.io/learn/softmax-activation/