BROKE: BERT-based Tokenized EULA Classifier
Solution Description Document

Contact information for Official Representative

Name: Chutian ("Chu") Weng
Email: chutianweng@gmail.com or chutianw@andrew.cmu.edu
Team Name: BROKE

Names of additional team members

No additional team members were involved.

Introduction to Team

Chu Weng is an undergraduate student at Carnegie Mellon University seeking to graduate with a B.S. in Business Administration and a B.S. in Statistics and Machine Learning. His primary areas of interest in machine learning are time series classification (TSC) and financial time series forecasting, though some of the information and insights from these fields are (somewhat) transferrable to NLP. When not at school, he lives in Maryland and enjoys running, going out with his friends (pre-covid), and ingesting copious amounts of protein powder while daydreaming about becoming a pro sprinter one day. Or making it to the Formula 1 grid, which would also be pretty cool, except he still hasn't figured out how to reverse park a normal car.

Executive Summary of Solution:

The proposed solution comprises an end-to-end deep learning and natural language processing (NLP) pipeline with a flexible user interface and flexible computational/storage requirements. The deep NLP model is based on BERT (Devlin et al., 2018) and is named BERT-based Tokenized

EULA Classifier (BROKE).[1] BROKE incorporates a publicly available pre-trained BERT model on top of convolutional downsampling layers and a novel set of memory modules, the architecture of which was partially inspired by dense convolutional networks (Huang and Liu et al., 2018). To increase flexibility and accessibility, the base BERT model was not modified (i.e., trained or otherwise changed), meaning that users with greater computational restrictions can easily swap the BERT model for a smaller or larger one.[2] Key highlights of the solution include

- the ability to use any number of PDF or Word documents;
- usage of nearly all primary and secondary data provided to train (and pre-train) BROKE, resulting in an F1 score of 0.589 and a Brier score of 0.118;[3]
- the ability to retrain BROKE based on relabeled data;
- an intuitive command-line interface that requires minimal user input and utilizes a clean file format;
- increased security by operating entirely offline and avoiding any non-PDF or non-Word documents so that contracts containing sensitive information can still be parsed;
- and flexible storage and computational requirements for increased accessibility.

BROKE Architecture: Technology Scope

BROKE and the associated document reading and parsing code are written entirely in Python. More specifically, BROKE was built and trained using the TensorFlow and Keras frameworks (Abadi et al., 2016; Chollet et al., 2015). A supplemental list of all required packages can be found in the `README` file in the `SUBMISSION` folder.

BROKE Architecture: Functionality and User Interface

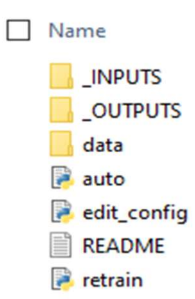
BROKE operates in a file format (see Fig. 1 below) and is accessible to the user via the command line. Inputs, in either Word or PDF format (or both), are placed into the `_INPUTS` folder. `auto.py` is then run, which, as the name suggests, requires no additional user input unless

[1] I know "BROKE" isn't the best acronym, but as an undergrad desperately looking for some work-study opportunities, this acronym deeply resonated with me.

[2] See https://github.com/google-research/bert for details; smaller models must have the same hidden state shape as the model used in training.

[3] On 700 previously unseen, randomly selected representative samples from the provided training set. The training F1 and Brier scores are 0.957 and 0.005, respectively, though Fig. 17 shows no evidence of overfitting.

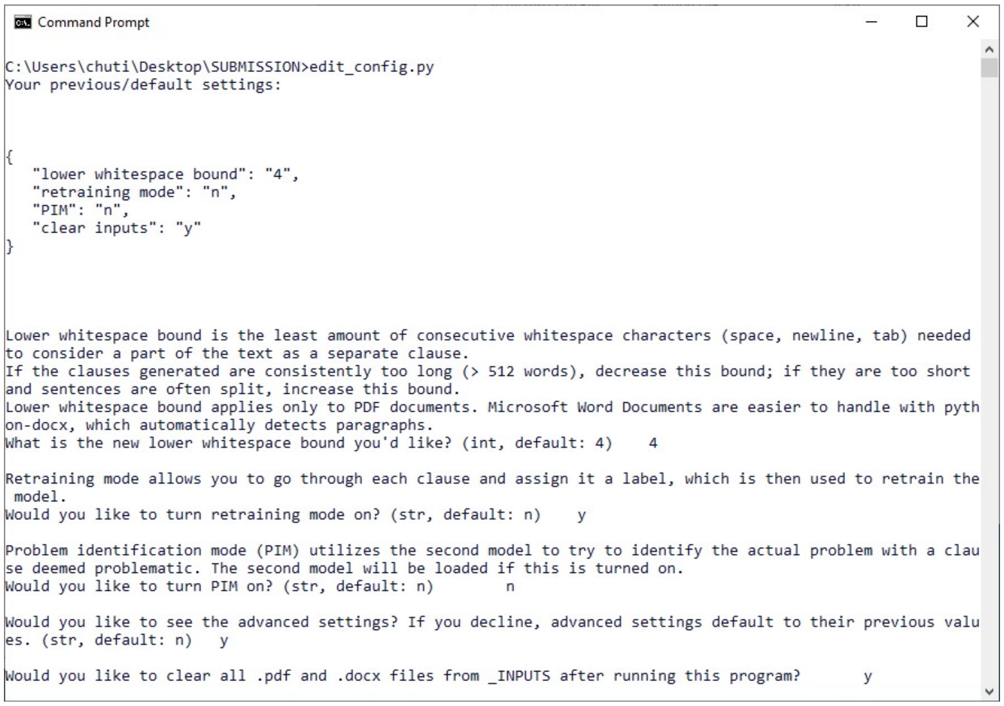
retraining mode is turned on (so that the user can add clause labels). The parsed and labeled clauses will be placed as CSV files in `_OUTPUTS`, and original file names are preserved.



```
Name
_INPUTS
_OUTPUTS
data
auto
edit_config
README
retrain
```

Fig. 1. The necessary files to run the program. Technically, the only directory that is strictly required is `data/models/broke.SavedModel`; `auto` and `edit_config` will create any missing directories.

If additional configuration is desired, the user can run `edit_config.py` to change clause spacing (used in parsing PDF documents), activate retraining mode, and determine whether the model should clear the inputs after labeling them. This user interface is extremely simple and essentially requires five to seven clicks (depending on whether dragging files into `_INPUTS` is considered a "click") to convert any number of Word and/or PDF documents to labeled clauses. The following figures provide a more detailed walkthrough of the process.



```
C:\Users\chuti\Desktop\SUBMISSION>edit_config.py
Your previous/default settings:



{
   "lower whitespace bound": "4",
   "retraining mode": "n",
   "PIM": "n",
   "clear inputs": "y"
}




Lower whitespace bound is the least amount of consecutive whitespace characters (space, newline, tab) needed
to consider a part of the text as a separate clause.
If the clauses generated are consistently too long (> 512 words), decrease this bound; if they are too short
and sentences are often split, increase this bound.
Lower whitespace bound applies only to PDF documents. Microsoft Word Documents are easier to handle with pyth
on-docx, which automatically detects paragraphs.
What is the new lower whitespace bound you'd like? (int, default: 4)    4

Retraining mode allows you to go through each clause and assign it a label, which is then used to retrain the
 model.
Would you like to turn retraining mode on? (str, default: n)    y

Problem identification mode (PIM) utilizes the second model to try to identify the actual problem with a clau
se deemed problematic. The second model will be loaded if this is turned on.
Would you like to turn PIM on? (str, default: n)        n

Would you like to see the advanced settings? If you decline, advanced settings default to their previous valu
es. (str, default: n)   y

Would you like to clear all .pdf and .docx files from _INPUTS after running this program?      y
```

Fig. 2. Running `edit_config.py` to enable retraining mode, since it is turned off by default.

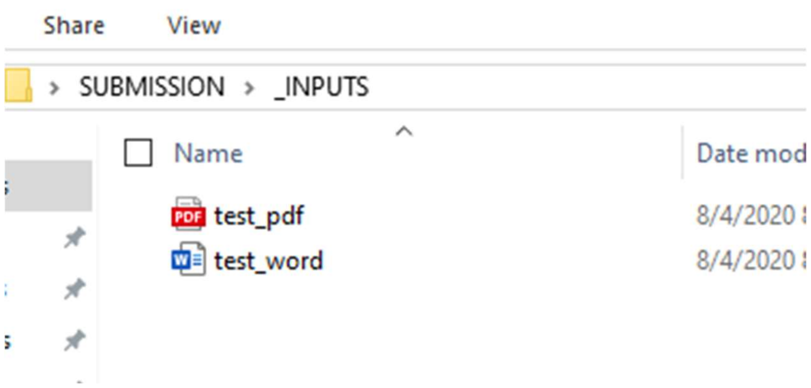
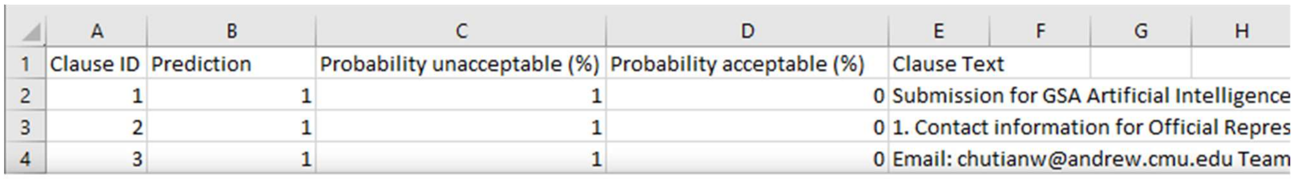
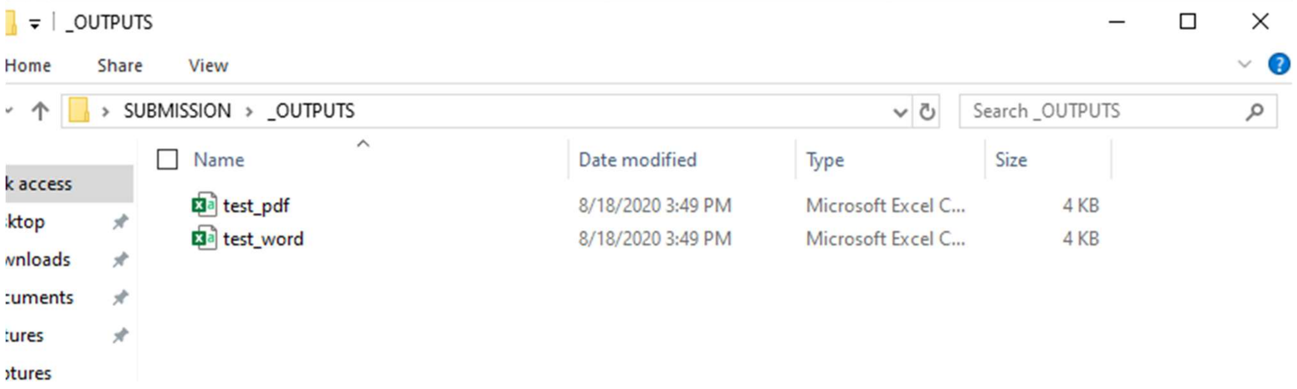


Fig. 3. Placing a document in two different formats into `_INPUTS`. This document is an earlier version of the Solution Description Document with significantly less text; I use this instead of an actual EULA so that I can demonstrate the clause labeling process for retraining mode without actually having to label hundreds of clauses.

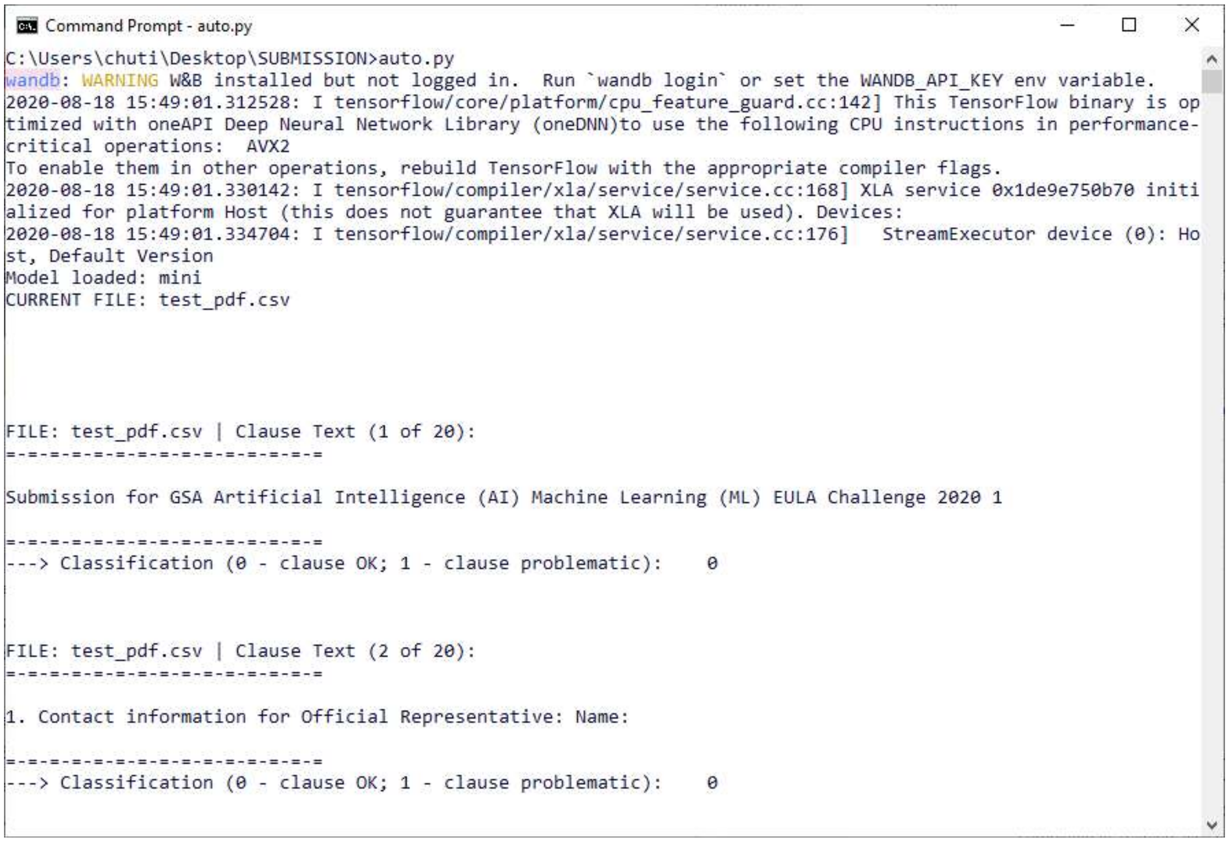


	A	B	C	D	E	F	G	H
1	Clause ID	Prediction	Probability unacceptable (%)	Probability acceptable (%)	Clause Text			
2	1	1	1	0	Submission for GSA Artificial Intelligence			
3	2	1	1	0	1. Contact information for Official Repres			
4	3	1	1	0	Email: chutianw@andrew.cmu.edu Team			



Name	Date modified	Type	Size
test_pdf	8/18/2020 3:49 PM	Microsoft Excel C...	4 KB
test_word	8/18/2020 3:49 PM	Microsoft Excel C...	4 KB

Fig. 4. Below: the state of the `_OUTPUTS` folder after running `auto.py`, with the original files converted to CSV format. Above: an example of one of the CSV files opened in Excel, though it can also be opened using any text editor or pandas if using a Jupyter Notebook. Please note that I was using a test model here, as the final BROKE model had not completed its training, which is why the probabilities are so extreme. Normally, they will be between 0 and 1, as seen in the Validation Data File.

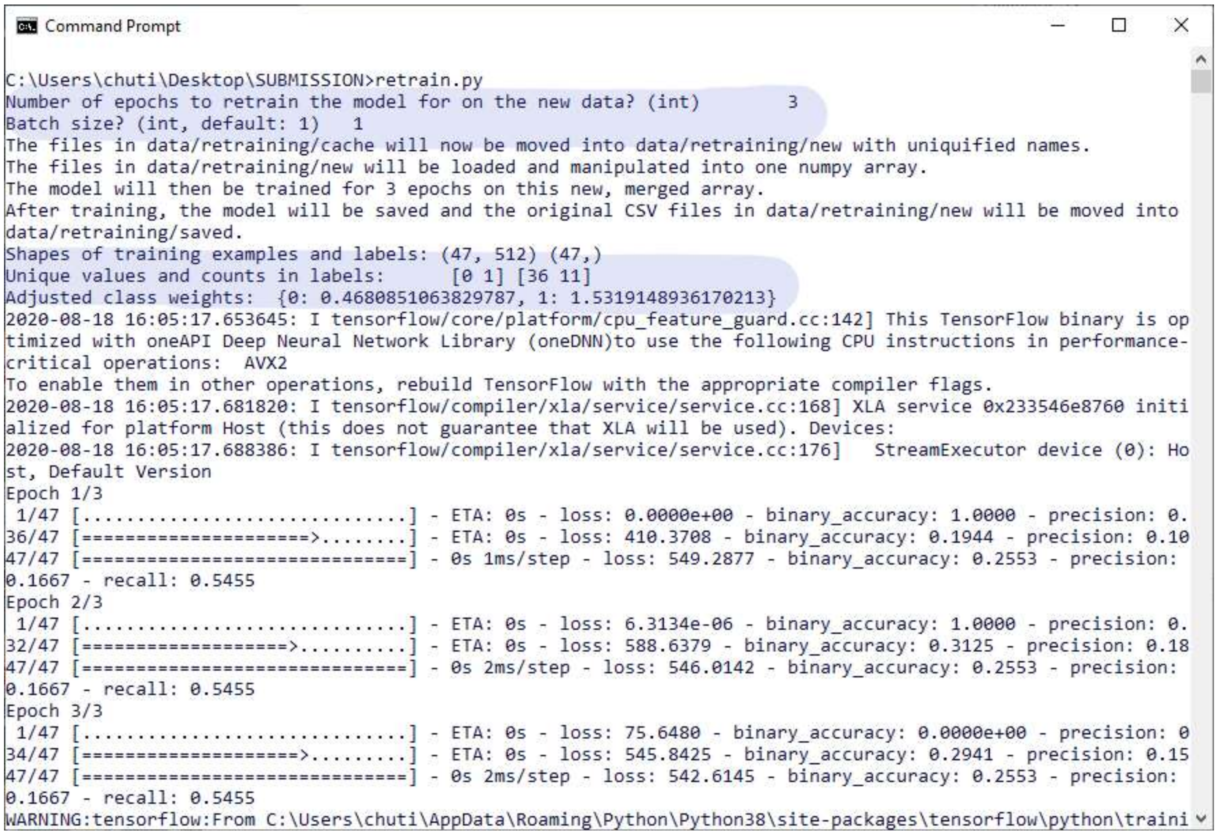


```
C:\Users\chuti\Desktop\SUBMISSION>auto.py
wandb: WARNING W&B installed but not logged in.  Run `wandb login` or set the WANDB_API_KEY env variable.
2020-08-18 15:49:01.312528: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is op
timized with oneAPI Deep Neural Network Library (oneDNN)to use the following CPU instructions in performance-
critical operations:  AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2020-08-18 15:49:01.330142: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x1de9e750b70 initi
alized for platform Host (this does not guarantee that XLA will be used). Devices:
2020-08-18 15:49:01.334704: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Ho
st, Default Version
Model loaded: mini
CURRENT FILE: test_pdf.csv




FILE: test_pdf.csv | Clause Text (1 of 20):
=-=-=-=-=-=-=-=-=-=-=-=-=-=

Submission for GSA Artificial Intelligence (AI) Machine Learning (ML) EULA Challenge 2020 1

=-=-=-=-=-=-=-=-=-=-=-=-=-=
---> Classification (0 - clause OK; 1 - clause problematic):    0


FILE: test_pdf.csv | Clause Text (2 of 20):
=-=-=-=-=-=-=-=-=-=-=-=-=-=

1. Contact information for Official Representative: Name:

=-=-=-=-=-=-=-=-=-=-=-=-=-=
---> Classification (0 - clause OK; 1 - clause problematic):    0
```

Fig. 5. `auto.py` is run in line 1, which produces the results seen in Fig. 4. Because retraining mode is enabled, additional user input is sought (in the bottom half of this figure). After providing labels, the user is asked whether they wish to retrain immediately or wait and independently execute `retrain.py` to retrain the model. At this point, the program will automatically save all labels in case of an emergency, so any sudden force-close has no consequences.

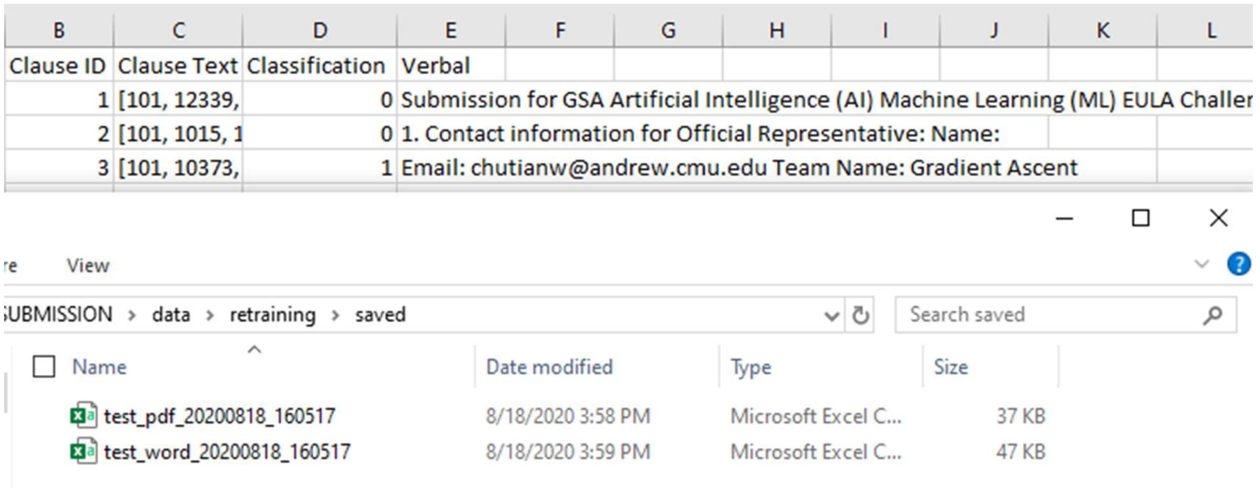
Here, please note that the same results can be obtained by double-clicking on `auto.py` rather than running it through the command line. If additional user input is required, as is the case for retraining mode, a command prompt or terminal window will automatically open. A custom window was used here to only preserve the black text on white background to make it easier to read alongside this document.

It is also important to note here that the next figure has a long caption, meaning that it will not fit onto this page; if this paragraph were not here, there would be a large white space between this page and the next, which would be quite awkward. I will take this opportunity to say that `edit_config`, `auto`, and `retrain` have line counts of only 75, 407, and 97, respectively, which is intentional—because downloading a set of files always presents an inherent security risk, the relatively few lines of code can easily be read over to ensure that they operate as expected.



```
Command Prompt

C:\Users\chuti\Desktop\SUBMISSION>retrain.py
Number of epochs to retrain the model for on the new data? (int)        3
Batch size? (int, default: 1)    1
The files in data/retraining/cache will now be moved into data/retraining/new with uniquified names.
The files in data/retraining/new will be loaded and manipulated into one numpy array.
The model will then be trained for 3 epochs on this new, merged array.
After training, the model will be saved and the original CSV files in data/retraining/new will be moved into
data/retraining/saved.
Shapes of training examples and labels: (47, 512) (47,)
Unique values and counts in labels:      [0 1] [36 11]
Adjusted class weights:  {0: 0.4680851063829787, 1: 1.5319148936170213}
2020-08-18 16:05:17.653645: I tensorflow/core/platform/cpu_feature_guard.cc:142] This TensorFlow binary is op
timized with oneAPI Deep Neural Network Library (oneDNN)to use the following CPU instructions in performance-
critical operations:  AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2020-08-18 16:05:17.681820: I tensorflow/compiler/xla/service/service.cc:168] XLA service 0x233546e8760 initi
alized for platform Host (this does not guarantee that XLA will be used). Devices:
2020-08-18 16:05:17.688386: I tensorflow/compiler/xla/service/service.cc:176]   StreamExecutor device (0): Ho
st, Default Version
Epoch 1/3
 1/47 [..............................] - ETA: 0s - loss: 0.0000e+00 - binary_accuracy: 1.0000 - precision: 0.
36/47 [=====================>........] - ETA: 0s - loss: 410.3708 - binary_accuracy: 0.1944 - precision: 0.10
47/47 [==============================] - 0s 1ms/step - loss: 549.2877 - binary_accuracy: 0.2553 - precision:
0.1667 - recall: 0.5455
Epoch 2/3
 1/47 [..............................] - ETA: 0s - loss: 6.3134e-06 - binary_accuracy: 1.0000 - precision: 0.
32/47 [===================>..........] - ETA: 0s - loss: 588.6379 - binary_accuracy: 0.3125 - precision: 0.18
47/47 [==============================] - 0s 2ms/step - loss: 546.0142 - binary_accuracy: 0.2553 - precision:
0.1667 - recall: 0.5455
Epoch 3/3
 1/47 [..............................] - ETA: 0s - loss: 75.6480 - binary_accuracy: 0.0000e+00 - precision: 0
34/47 [====================>.........] - ETA: 0s - loss: 545.8425 - binary_accuracy: 0.2941 - precision: 0.15
47/47 [==============================] - 0s 2ms/step - loss: 542.6145 - binary_accuracy: 0.2553 - precision:
0.1667 - recall: 0.5455
WARNING:tensorflow:From C:\Users\chuti\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\traini
```

Fig. 6. The output of `retrain.py`, which was run after providing clause labels during `auto.py`. The first highlighted region shows that the user can customize the number of epochs and batch size during the retraining process, and the second shows that class weights are automatically adjusted and applied to the data to address an imbalance in the relative proportions of the labels. Again, this is a test model used for quick prototyping, which is why the model achieves a maximum loss equivalent to the number of pounds I wish I could bench press.



B	C	D	E	F	G	H	I	J	K	L
Clause ID	Clause Text	Classification	Verbal							
1	[101, 12339,	0	Submission for GSA Artificial Intelligence (AI) Machine Learning (ML) EULA Challer							
2	[101, 1015, 1	0	1. Contact information for Official Representative: Name:							
3	[101, 10373,	1	Email: chutianw@andrew.cmu.edu Team Name: Gradient Ascent							

Name	Date modified	Type	Size
test_pdf_20200818_160517	8/18/2020 3:58 PM	Microsoft Excel C...	37 KB
test_word_20200818_160517	8/18/2020 3:59 PM	Microsoft Excel C...	47 KB

Fig. 7. Below: the labeled data are placed into `data/retraining/saved` so the user can retrain the model further in the future if they wish. Above: a labeled data CSV, opened in Excel. The "Clause Text" column is now the tokenized version of the "Verbal" column; this allows faster initialization for any future retraining.

BROKE Architecture: Application of Artificial Intelligence / Machine Learning

Please note: the demo video also presents mostly the same information as everything below, albeit in a more engaging format. I personally would prefer watching a presentation over reading a huge wall of text, so I thought it'd be nice to provide you with the option to read this or watch the second part of the demo (or both, if you like to hear my bad jokes)—whichever you prefer.

BROKE is a deep neural network built on top of a pre-trained BERT (Devlin et al., 2018) model, which itself was built using the Transformer architecture (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, 2017). Outputs from BERT are downsampled via convolutional layers that build upon the ideas presented in (Lin, Chen, and Yan, 2013). BROKE also incorporates novel Dense Static Memory Resolution (DSMR) and Adaptive Memory Resolution (AMR) modules[4] to help overcome the challenges posed by the limited amount of training data. Each of the 14 DSMR modules hold their own internal memory tensor, which is compared to the input text. The internal memory tensors are static and not trainable for the DSMR modules; for the AMR module, however, the memory tensor *is* trainable, with the intent to allow the model to store its own representation of a "rule" to determine if a clause is acceptable or not. The outputs from these modules are concatenated with a pooled version of the original BERT output and fed into a standard densely/fully connected feedforward network with a sigmoid classifier.

BERT Overview

BERT (Devlin et al., 2018) consists of the encoder portion of a Transformer (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, 2017) pre-trained on 3.3 billion English words in their natural contexts.

[4] Three days before this document is due, I conducted a final literature review to make sure this had not been done already, and I found (Weston, Chopra, and Bordes, 2014)'s MemNN, which employs an adaptive-memory-like component (admittedly, a much more detailed one) for question answering (QA) tasks. DSMR/AMR modules were formulated independently of this paper, a fact made most apparent by MemNN's detailed incorporation of "slotting" new memory in the network as opposed to BROKE's utilization of majority-static (14/15 modules) memory in isolated modules. Still, despite the fact that this paper was not referenced in the creation of BROKE, I have included it in the references anyways to acknowledge its contribution to the field as a whole.

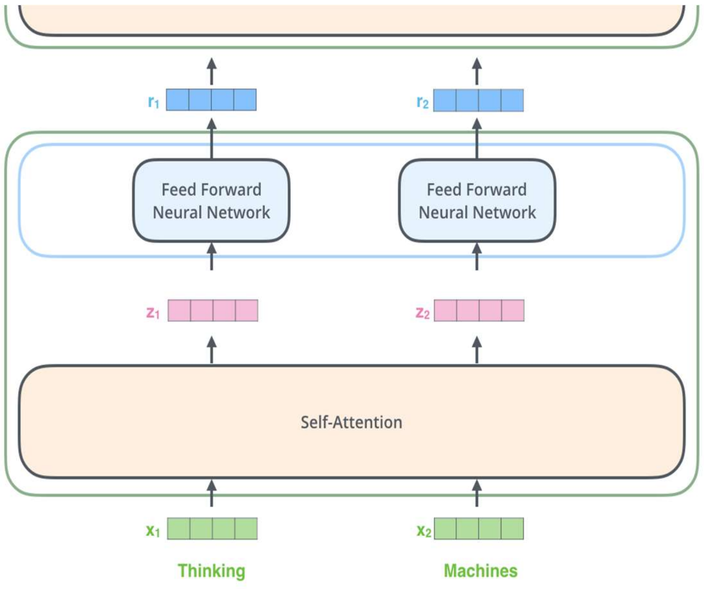


Fig. 8. A detailed look into the encoder portion of a Transformer. *Source: The Illustrated Transformer by Jay Alammar, 2018. http://jalammar.github.io/illustrated-transformer/*

First, Transformers offer significant benefits over LSTM (Hochreiter and Schmidhuber, 1997) and other recurrent architectures. Recurrent neural nets (RNNs) are notoriously difficult to train, both in terms of training time and gradient backpropagation over very long sequences. Rather than processing input sentences sequentially, Transformers process each input embedding simultaneously, allowing for increased contextual awareness, especially when combined with a self-attention mechanism, which allows words in an input sentence to be contextualized relative to each other.

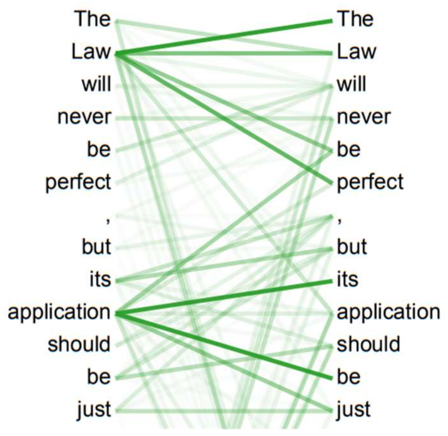


Fig. 9. Self-attention visualized on a sample sentence. *Source: Attention is all you need (Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, 2017)*

Thus, BERT, which utilizes this encoder portion of a Transformer, is able to learn language structure and context much faster and to a more in-depth extent than RNN-based models.

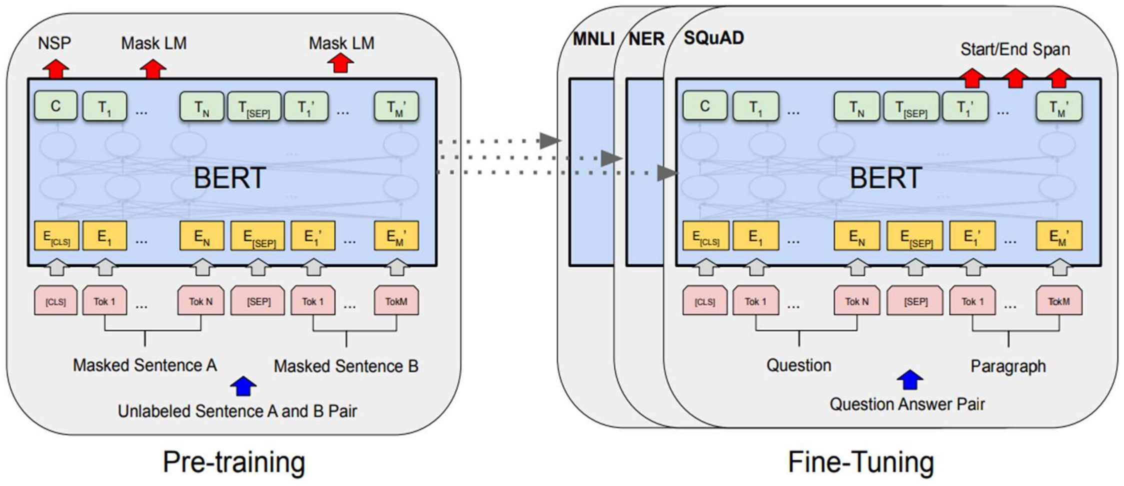
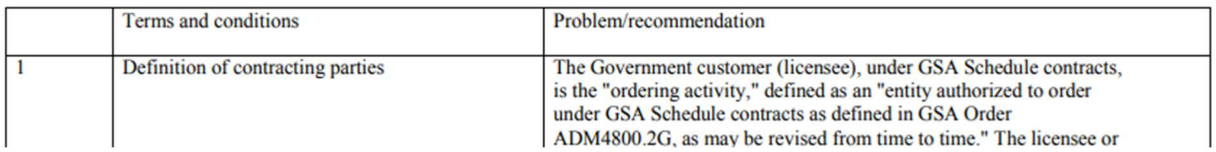


Fig. 10. BERT's pre-training and fine-tuning processes in detail as presented in the original paper. *Source: BERT: Pre-training of deep bidirectional transformers for language understanding (Devlin et al., 2018)*

BERT was pre-trained through next sentence prediction (NSP) and masked language modeling (MLM) tasks, which are both unsupervised. In NSP, pairs of sentences are passed in and the model predicts whether the second sentence follows the first or not. In MLM, words are randomly masked before the sentence is passed into the model, and the model predicts the identities of the masked words.

Memory Layers

The motivation for these layers is the observation that a "cheat sheet" like the provided Appendix B is very helpful in identifying potentially problematic clauses (see Figure 11 below). With a sufficiently-sized training set, incorporating the Appendix B information would be unnecessary; however, as there is limited training data, allowing the model to access this information could prove helpful by providing additional knowledge, especially in the first stages of training. In many ways, the implementation and goals of these memory modules is akin to those of transfer learning—information gathered outside of the training set is made accessible in the form of parameters as the network is further fine-tuned on the training data.



	Terms and conditions	Problem/recommendation
1	Definition of contracting parties	The Government customer (licensee), under GSA Schedule contracts, is the "ordering activity," defined as an "entity authorized to order under GSA Schedule contracts as defined in GSA Order ADM4800.2G, as may be revised from time to time." The licensee or

Fig. 11. An example of information from Appendix B that may be useful in identifying problematic clauses. *Source: Appendix B, Government Services Administration (GSA) 2020 AI/ML EULA Challenge https://github.com/GSA/ai-ml-challenge-2020/tree/master/reference*.

One significant difference between memory modules and transfer learning, however, is that transfer learning incorporates this known information via existing sets of parameters (i.e., layers) that are fine-tuned or frozen and built upon, while memory modules incorporate this information by adding *additional* parameters. The usage of additional parameters which augment the network graph could potentially offer an advantage over transfer learning approaches in this task by separating the gradient flow of new and known information. In other words, when fine-tuning a transfer-learning-based model, the gradient of the loss on the training data flows directly through the parameters which encode the information learned from the model's pre-training data. However, the proposed memory modules separate this gradient flow such that there exists a direct path from the training inputs through the model that is independent of the known data, allowing the model significantly more flexibility in determining which aspects, if any, of the known information are applicable to the new task.[5] [6]

[5] While this applies more to fine-tuning processes where all pre-trained parameters are still trainable, it is also true for transfer learning processes where the learned pre-trained weights are frozen. In essence, transfer learning forces new information through the lens of previously learned tasks, while memory modules allow significantly greater flexibility for the model to "discover" the new data itself while still having access to the known information. A very poor implementation of transfer learning can hurt a model's performance (initially for fully retrainable models and permanently for partially frozen models); it's generally impossible for memory modules to do the same.

[6] I'm not against transfer learning—I used it as well in the hopes that it would "warm up" the weights—but I did want to point out the advantages of these memory modules on this particular task, especially because coding them took me *so* many trips to Stack Overflow.

How the GSA Appendix B information is incorporated into the model is relatively straightforward. For each of the 14 rows, the text in the first column ("Terms and conditions") is concatenated with the text in the second column ("Problem/recommendation"), delimited by a colon. This concatenated string is then tokenized, padded, and fed into a pre-trained BERT, which outputs a 512x768 matrix. This serves as the memory matrix stored by each of the 14 Dense Static Memory Resolution (DSMR) modules, one for each row of the provided Appendix B.

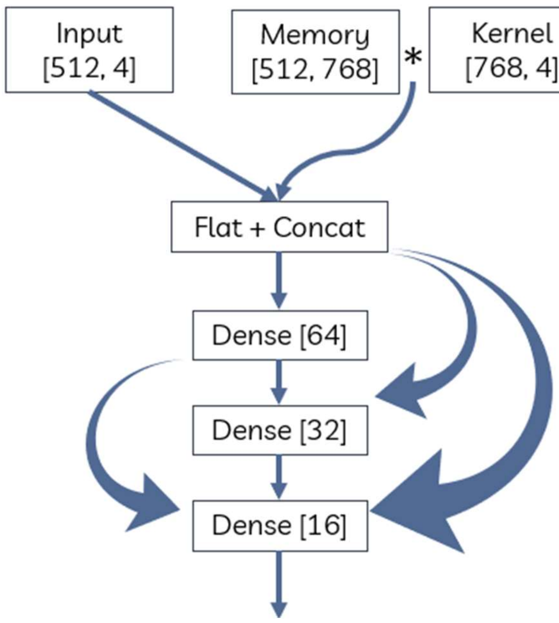


Fig. 12. A schematic representation of DSMR modules with output shapes given in brackets. Here, the asterisk operator A*B denotes the matrix multiplication of A and B, and not convolution.

The module also stores additional parameters in the form of a learnable kernel used to reduce the dimensionality of the memory matrix. When the module is called on an input, it flattens the input. Then, it calculates the matrix product of its memory and kernel, adds the bias, and applies a nonlinear activation function.[7] The resulting matrix is then flattened and concatenated onto the flattened input matrix, which is then fed through three densely connected layers.[8] One difference here is that not only are the neurons

[7] Within the memory modules, PReLU (He et al., 2015) was used for additional flexibility through increased parameters.

[8] In the interest of time and conciseness, the mathematical description of these memory modules has been assigned to Appendix A below.

densely connected, but the layers themselves are as well, a feature inspired by DenseNet (Huang and Liu et al., 2018). This design is the result of the observation that the input to the memory modules is a downsampled BERT output, meaning that it has already passed through several convolutional layers. Furthermore, the outputs from the memory modules are passed through more dense layers. To avoid weak or overly strong gradients in backpropagation, each dense layer within the memory module is connected to the input, facilitating gradient flow.

The architecture of the Adaptive Memory Resolution (AMR) modules is similar to that of the DSMR modules, with the exception being that the memory matrix is trainable and the modules do not have kernels,[9] with the intention to allow the module to learn its own representation of a general "rule" to compare the input to.

BROKE Architecture and Implementation

BROKE was pre-trained on a portion of the additional SRT data[10] before moving on to a supervised learning process. During the training phase, inputs of variable-length strings were preprocessed for special characters before being tokenized. The tokenizer added BERT's special `[CLS]` and `[SEP]` tokens before padding each clause to the maximum length of 512 tokens. For clauses which were too long, they were either split in a sliding-window fashion if the label was "0" or discarded if the label was "1".[11] Dropout (Srivastava et al., 2014) was incorporated into many layers (see network graph below); the dropout rate for all layers using dropout was 0.1. For non-memory layers, the activation function used was the Swish function $Swish(x) = x \cdot \sigma(\beta x)$, where β is either a constant or a trainable parameter and σ denotes the sigmoid function $\sigma(x) = \frac{1}{1+e^{(-x)}}$ (Ramachandran, Zoph, and Le, 2017). In the TensorFlow implementation, β = 1.

[9] Since the memory matrix is linearly combined with the kernel and bias, having a trainable kernel and bias on top of a trainable memory matrix would be unnecessary.

[10] Many of the words in the SRT data were cut off, and many clauses were exceptionally long. While cutting clauses is possible for "0" labels as any subset retains the same label, this is not necessarily true for "1" clauses. I entered this challenge a bit late and decided to just select the best subset rather than spend more time preprocessing the SRT data.

[11] In the interest of time—which I had very little of—I couldn't read the entire thing if it had a "1" label and try to figure out which subclause was problematic. Thankfully, there were very few of these cases.

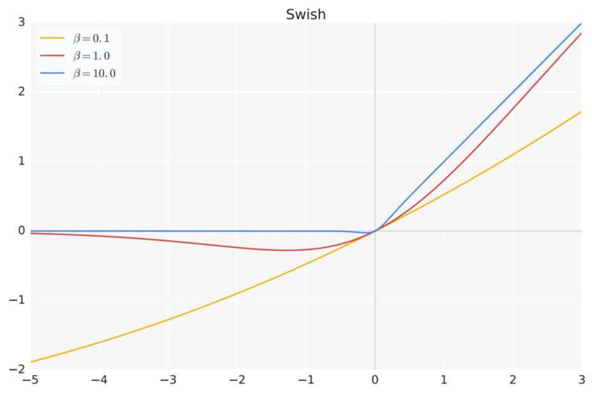


Fig. 13. A graph of the Swish activation function for various β. In BROKE, β = 1, corresponding to the red (middle) line.
Source: Searching for activation functions (Ramachandran, Zoph, and Le, 2017)

Inputs of shape (512) are passed into BERT, which outputs a (512, 768) matrix and a (768) pooled version of the matrix, which are referred to as `BERT[0]` and `BERT[1]`, respectively. `BERT[0]` is then fed into a series of three convolutional downsampling layers with output shapes (512, 128), (512, 32), and (512, 4), respectively; all layers use a 1-wide convolution, following the ideas presented in (Lin, Chen, and Yan, 2013).

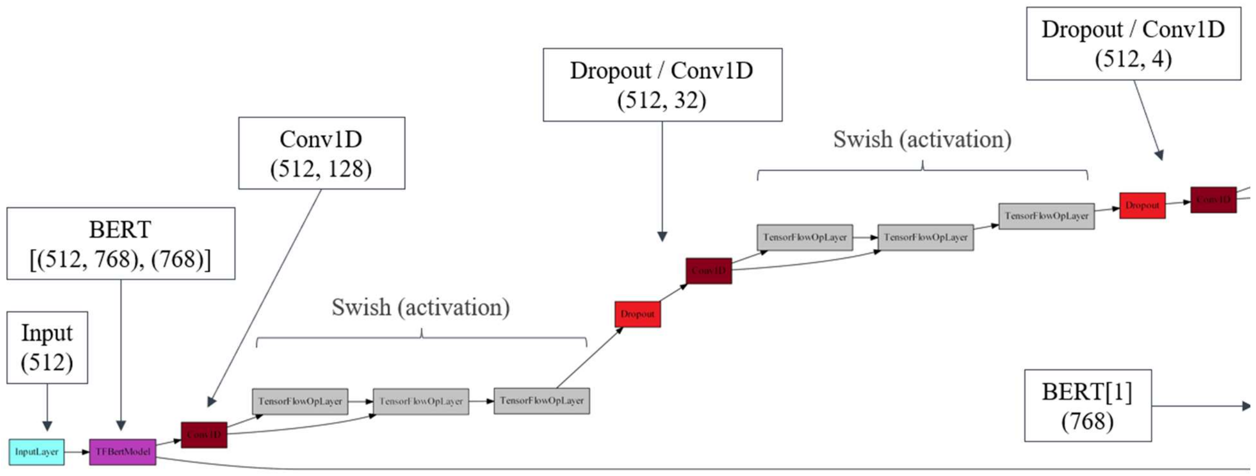


Fig. 14. A diagram of the first half of BROKE's architecture, as presented the video demonstration.

The downsampled (512, 4) matrix is then passed into 14 DSMR layers, 1 AMR layer, and a flattening layer. Each of the memory layers outputs a vector of shape (16), and the outputs from these 16 layers (15 memory + 1 flatten) are concatenated with `BERT[1]`, resulting in a (3056)-dimensional vector. This vector is passed through 4 dense layers of size (256), (256), (16), and (1), with the last layer using sigmoid activation.

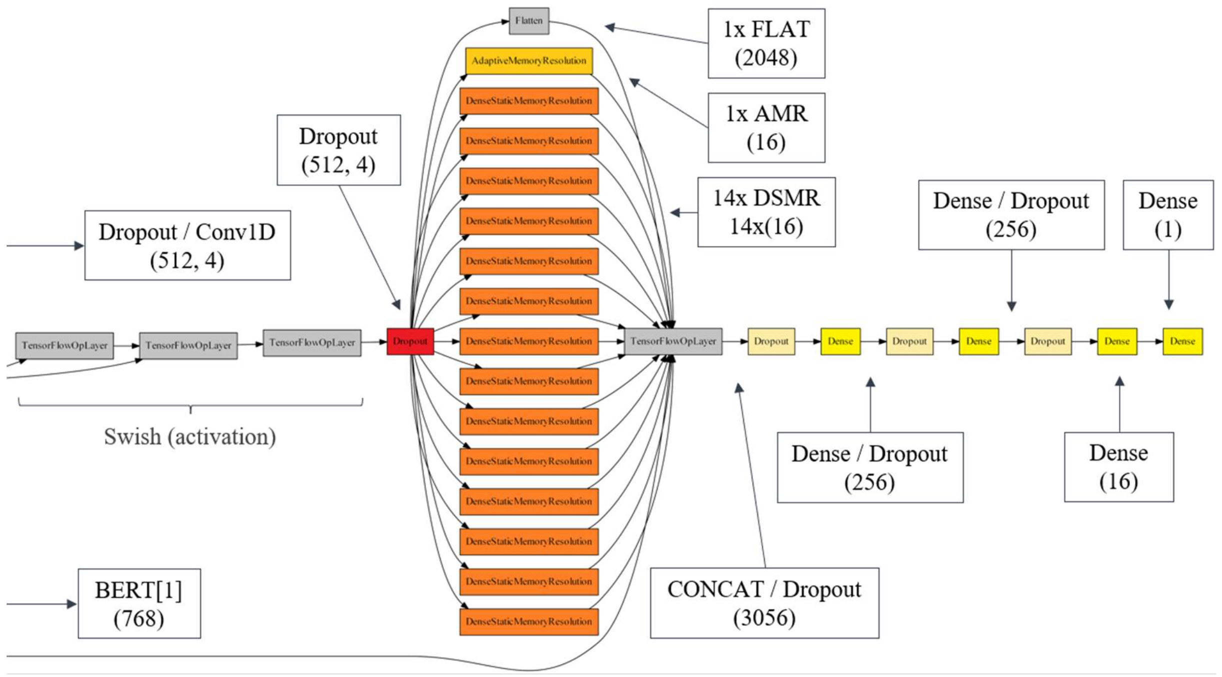
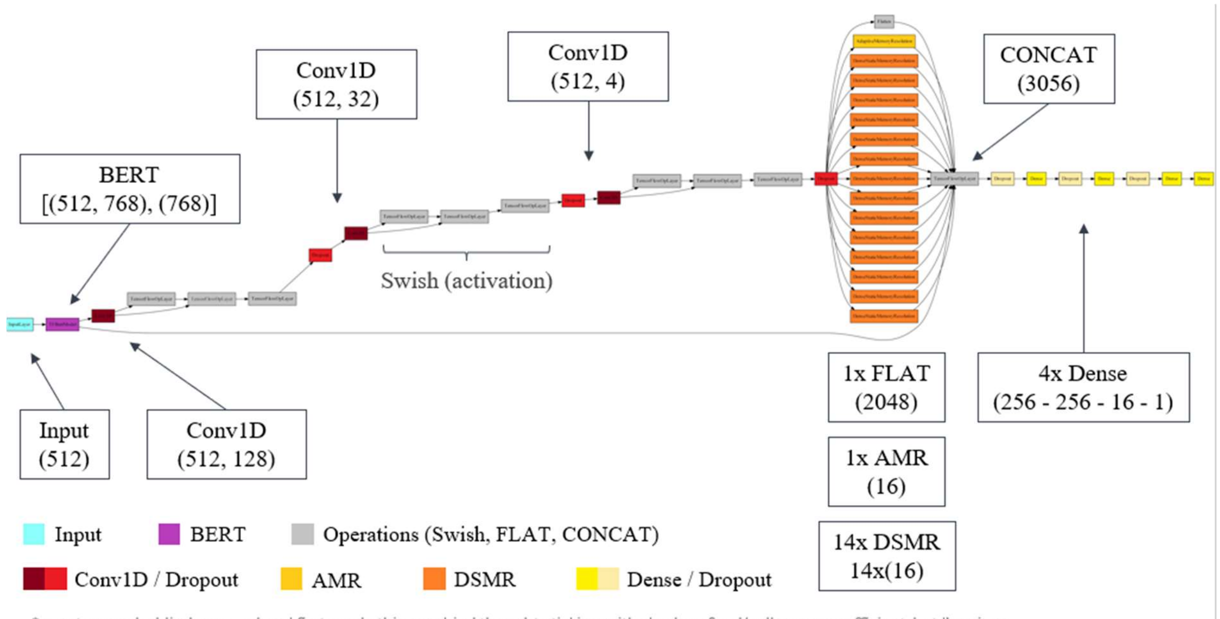


Fig. 15. The second half of BROKE's architecture, as presented the video demonstration.

Overall, the design philosophy for BROKE's architecture was to provide maximum flexibility, both in terms of customization (customizing which pre-trained BERT model to use for different storage/computational restrictions and/or adding additional memory layers)[12] and in terms of gradient flow. Because time was a major constraint in the creation and training of BROKE, opting for more flexibility allowed for rapid prototyping and training, thus avoiding extensive, time-consuming experimentation on the usefulness and behavior of every module and hyperparameter. As stated previously, since the training data can pass through the entire network unaffected by non-BERT known data (in the form of frozen pre-trained parameters or the memory held in DSMR modules), it is easy for the model to discard known information if necessary.

[12] For example, the memory for the DSMR layers are just BERT's representations of guidelines for certain types of unacceptable clauses in contracts. These existing guidelines are not comprehensive, so more guidelines (like "clauses that allow the company to steal money and donate it to a team that is not the Ravens should be deleted") can be typed out by the user in a string format that the user understands and passed through BERT, which returns a matrix representation. This matrix can serve as the memory matrix for a new DSMR layer which could be incorporated into the model, which would be retrained knowing this information. This process, in effect, is like "talking" to the model.



* a note on colorblindness – when I first made this graphic, I thought sticking with shades of red/yellow was sufficient, but I've since done more research and found that this image probably looks horrendous for those with deuteranopia and protanopia. It's too late (<24 hours until submission) to change this image, but I do apologize, and I redid the loss graphs later using a custom color palette.

Fig. 16. The full architecture of BROKE.

By being fully transparent in its architecture and code and choosing to remain entirely offline, BROKE provides significant benefits in both security and customization. The user's ability to run BROKE depends only on the intactness of the files and nothing else. Malicious executables, even if placed as input files, are ignored. Also, I'm only (about to be) a first-year undergrad, so two major constraints were time and computing resources, which also suggests that the full potential of BROKE's architecture has not been exploited. This can be seen most clearly in the graph of the validation metrics:

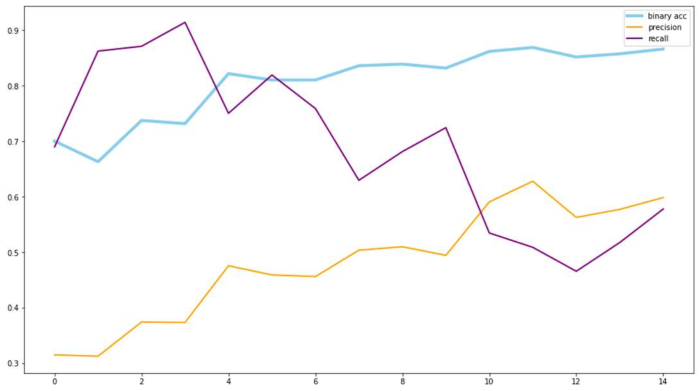


Fig. 17. Binary accuracy (blue), precision (yellow), and recall (purple) on the validation set for the first 14 epochs. Please note how the precision and recall had just converged (possibly due to a poor initialization) and are both increasing, as is the binary accuracy.

The hyperparameters and other data are summarized as follows:

- optimizer: Adam (Kingma and Ba, 2014) with learning rate 1e-5
- batch size (epochs): 1 (9), 2 (1), 1 (3), 4 (4) (Smith and Kindermans et al., 2018)
- dropout rate: 0.1 for all layers
- class weights: {0: 0.36, 1: 1.64}
- final training metrics (7344 ex.): 0.98 acc., 0.92 prec., 0.99 rec., 0.957 F1, 0.005 Brier
- test set metrics (700 ex.): 0.874 acc., 0.643 prec., 0.543 rec., 0.589 F1, 0.118 Brier
- visits to Stack Overflow: way too many

References

[Abadi et al., 2016] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In the *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265-283, 2016.

[Chollet et al., 2015] François Chollet et al. Keras. URL `keras.io`.

[Devlin et al., 2018] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. 2018. URL `arxiv.org/abs/1810.04805`.

[He et al., 2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. 2015. URL `arxiv.org/abs/1502.01852`.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735-1780, 1997.

[Huang and Liu et al., 2016] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[Kingma and Ba, 2014] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *The International Conference on Learning Representations (ICLR)*, 2014.

[Lin, Chen, and Yan, 2013] Min Lin, Qiang Chen, Shuicheng Yan. Network in network. 2013. URL `arxiv.org/abs/1312.4400`.

[Ramachandran, Zoph, and Le, 2017] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. 2017. URL `arxiv.org/abs/1710.05941`.

[Smith and Kindermans et al., 2018] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In the *International Conference on Learning Representations (ICLR)*, 2018.

[Srivastava et al., 2014] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929-1958, 2014.

[Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 6000-6010, 2017.

[Weston, Chopra, and Bordes, 2014] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. 2014. URL `arxiv.org/abs/1410.3916`.

Appendix

Appendix A: Mathematical Description of DSMR/AMR Modules

Showing how the memory modules are coded is much easier through a diagram than mathematically, but I thought I would include a mathematical description here for completeness. I'll be honest, I'm not a huge math guy—I've never written a mathematical proof or anything like that before—so please excuse me if I deviate from any established conventions that I'm not aware of. Seriously, I just got out of high school a few months ago and it's literally my first time, so please don't laugh!

Definitions and notation. We first define a concatenation operation and notation for any two arbitrary matrices or vectors $A \in \mathbb{R}^{a \times b}$ and $B \in \mathbb{R}^{a \times c}$ to be $[A\!:\!B] \in \mathbb{R}^{a \times b + c}$, such that the property $[CA\!:\!CB] = C[A\!:\!B]$ holds true for any third arbitrary matrix $C \in \mathbb{R}^{d \times a}$.

We also define a double-iterated composition with concatenation operator: $[\circledcirc]_N^f(x) = f_N([x\!:\!f_{N-1}([x\!:\!f_{N-2}([x\!:\!\ldots f_1(x)])])\!:\!f_{N-2}([x\!:\!f_{N-3}([x\!:\!\ldots f_1(x)])])\!:\!\ldots\!:\!f_1(x)])$. This is "double-iterated" for a set of sequential functions f because $f_n,\ n \in 1, 2, 3 \ldots N$ takes as input the concatenated output of every previous function and the original input (i.e., the structure of the above equation is $[\circledcirc]_N^f(x) = f_N([f_{N-1}(\ldots)\!:\!f_{N-2}(\ldots)\!:\!\ldots\!:\!f_1(x)\!:\!x])$), which implies every function prior also does the same. Thus, if $SHAPE(x)$ returns the dimensionality of a vector (i.e., if $x \in \mathbb{R}^k$, $SHAPE(x) = k$) and $SHAPE(f_n(.)) = k_n$, it follows that $f_n\!:\ \mathbb{R}^s \to \mathbb{R}^{k_n}$ where $s = SHAPE(x) + \sum_{i=1}^{n-1} k_i$, with k being a positive integer.

Finally, we define a flatten operation for an arbitrary tensor x: $FLAT(x),\ x \in \mathbb{R}^{a_1 \times a_2 \times \ldots \times a_n}$, where the flatten operation reshapes x deterministically while preserving the entries of x such that $FLAT(x) \in \mathbb{R}^A,\ A = \prod_i a_i$.

DSMR formulation. A dense layer of a neural network can be treated as a nonlinear function $D_n(.),\ n \in 1, 2, 3 \ldots N$, where the subscript indicates the layer number within the memory module. The dense layers of the memory modules are represented with this notation; a function D_n outputs a vector of k_n dimensions.

We seek to compare an input matrix $I \in \mathbb{R}^{t_I \times d_I}$ and an immutable memory matrix $M \in \mathbb{R}^{t_M \times d_M}$ in some learned manner such that the comparison produces a comparison feature vector $V \in \mathbb{R}^{k_N}$.

First, we initialize a kernel matrix $K \in \mathbb{R}^{d_M \times d_K}$, which allows for a learned representation of the memory matrix. In practice, d_K is selected such that $d_K < d_M$, which has the added benefit of reducing the number of entries in the matrix product MK, i.e. $SHAPE(FLAT(MK)) < SHAPE(FLAT(M))$. With the definitions above, the formulation of a DSMR layer is relatively straightforward:

$$DSMR(I, M) = [\circledcirc]_N^D([FLAT(I): FLAT(MK)])$$

AMR formulation. Similarly, borrowing the notation above, an AMR layer is almost identical except the memory matrix is trainable, which also means that a kernel is no longer required. Thus, an AMR with N dense layers D is defined as follows:

$$AMR(I, M) = [\circledcirc]_N^D([FLAT(I): FLAT(M)])$$

The concatenation and flattening in both the DSMR and AMR equations could be shortened from $FLAT(I): FLAT(MK)$ to $FLAT(I: MK)$ for DSMR layers or from $FLAT(I): FLAT(M)$ to $FLAT(I: M)$ for AMR layers in the case where $t_I = t_M$, which is true for BROKE. However, this shortened version is not used both because it is less general and because, in practice, the matrices are flattened individually before concatenation to allow for a clear separation between the flattened input and the flattened memory (or memory-kernel product). While a distinct separation is completely irrelevant to the model itself, it may allow for easier future experimentation and interpretation for the end user.

In the BROKE model, the values are set as follows:

$$N = 3; \quad k_n = \frac{128}{2^n}$$

For DSMR layers specifically:

$$t_I = t_M = 512; \quad d_I = d_K = 4; \quad d_M = 768$$

and for AMR layers:

$$t_I = t_M = 512; \quad d_I = d_M = 4$$