Training A RNN for detecting Clickbait

Introduction

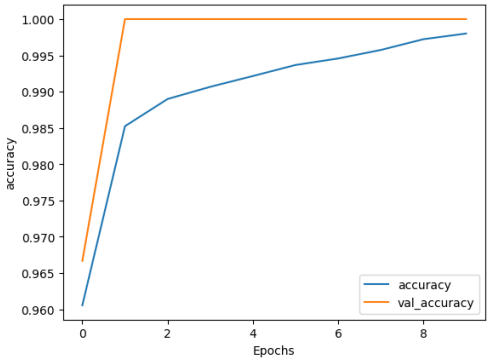
At the beginning of the module, I thought about how I could continue with my project ClickbaitML (a clickbait detection algorithm) and if I could use it for my assessment. ClickbaitML was a project I worked on for my Abitur (Germany's final high school exam). ClickbaitML was initially one of my *Jugend forscht*[1] projects but quickly gained importance in the media and clickbait landscape over the last few years. It was also one of my favorite projects because it was innovative and presented numerous challenges during development. The fascination for the project persisted, and when I discovered an English language clickbait dataset with over 32,000 entries on Kaggle[2], I knew I had to work on this for AI Basics. Surprisingly, the data split was also even. My goal for this module is to work with this dataset and a variety of Machine Learning tools to test how I could best detect clickbait and learn from the models.

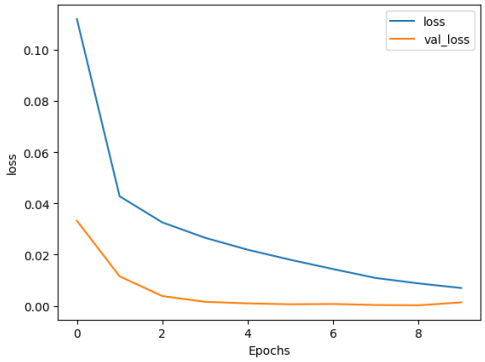
The First Model - Recursive Neural Networks (RNNs)

Fast forward, I created a Jupyter Notebook and trained a Recursive Neural Network (RNN). I chose this concept because it was described as promising by TensorFlow[3], and I wanted some good benchmarks on that dataset. So, I used the model description provided by the article and worked with that. Here is the model description:

Layer (type)	Output Shape	Param #
text_vectorization_1 (TextVectorization)	(2, 64)	0
embedding_2 (Embedding)	(2, 64, 64)	640,000
bidirectional_2 (Bidirectional)	(2, 128)	66,048
dense_4 (Dense)	(2, 64)	8,256
dense_5 (Dense)	(2, 1)	65

At first I installed some tensorflow extensions to enhence the performance on my m1 but realized that even with those extensions it took 8min per epoch wich is pretty long, since I wanted to change the neural network to experiment. So I remembered myself of working with google colab in the learning units and tried to run the code on a T4 GPU which was great to use since it needed under 5 min per epoch.The results after 10 epochs were pretty amazing! Not only did the validation accuracy rise to nearly one hundred percent, but the loss approached zero! (Visualized by the graphs below)





[1] https://www.jugend-forscht.de/

[2] https://www.kaggle.com/datasets/amananandrai/clickbait-dataset

[3] https://www.tensorflow.org/text/tutorials/text_classification_rnn

Testing the model then on the entire testing dataset revealed then a slightly less advenced score but was also awesome. The test loss was only 0.168 and the accuracy was around 97%.

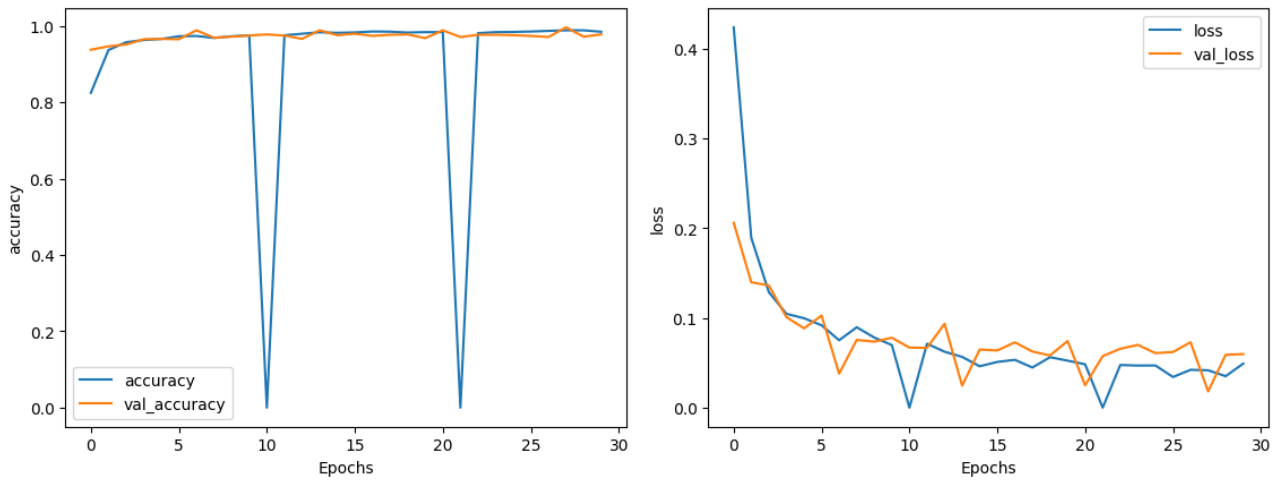
This was really a strong benchmark and wasn't easily beatable. So let's analyze what this classification RNN really consists of and then how to make it better (not only be more epochs) and what other ways we could use to predict clickbait.

At first we need to have a look at our data source, here are a few example rows:

headline	clickbait
Should I Get Bings	1
Which TV Female Friend Group Do You Belong In	1
The New "Star Wars: The Force Awakens" Trailer Is Here To Give You Chills	1
This Vine Of New York On "Celebrity Big Brother" Is Fucking Perfect	1
A Couple Did A Stunning Photo Shoot With Their Baby After Learning She Had An Inoperable Brain Tumor	1
How To Flirt With Queer Girls Without Making A Total Fool Of Yourself	1
32 Cute Things To Distract From Your Awkward Thanksgiving	1
If Disney Princesses Were From Florida	1
What's A Quote Or Lyric That Best Describes Your Depression	1
Natalie Dormer And Sam Claflin Play A Game To See How They'd Actually Last In "The Hunger Games"	1

The Data-Table consists of two columns: headlines of articles and ratings indicating if they are clickbait or not. After transferring this into train and test data sets, a text vectorization layer was created to process only the training data, avoiding handling unknown words from the test data. The maximum vocabulary was set to 1000 words, and the data was normalized by converting every word to lowercase and removing all punctuation. This layer was then included in the actual model (image on page 1) and connected to an embedding layer to transform words into vectors. Following these layers, the bidirectional layer enables the neural network to reuse past information to detect patterns in the inputs and learn how to detect clickbait. After this layer, two dense layers follow: one to process the output of the bidirectional layer and one single-node output layer. The first dense layer uses relu as an activation function to work with more complex nonlinear relations and learn more complex relations. The second one has sigmoid as an activation function to reliable return values between 0 and 1 and also minimize loss.For the training of the neural network, as mentioned, I used 10 epochs with a 0.2 test split, which equates to around 25,000 data points for training and approximately 7,000 for validation. The batch size was set to 128 because there is enough other data to learn from. The results as

Moving on and modifying the network, I want to make it able to find more reliable patterns even if the data is incomplete. For this I added a dropout layer and also minimized the data exposure per epoch so it has to adapt to a more divers set of data every epoch and has it harder to find good relations. For this I configured the dropout to be 0.4 and the steps per epoch down to 2560. The batchsize remains at 128 for now and I trained the model for 30 epochs. Here are the graphs:



The two train accuracy and loss drops happened because of a data fitting problem where he didn't find new data for the epoch. These can be ignored. What we can see for this training round is that the validation results at the beginning were quite worse than in the first round without the dropout layer and the new training strategy. But then, nears itself the graph we saw before and hits the same test results. After validating the entire test base, we now have only a loss of about 0.06 and an accuracy of 97.6%, wich is even better than before! I don't think the improved accuracy is caused by the dense layer, but rather by the increased epochs. What was also stunning was the new speed of the neural network, in the previos training the ten epochs took around 50min on the Google T4 GPU and the 30 epochs now took only 11! Thats a factor 5, with a better result!

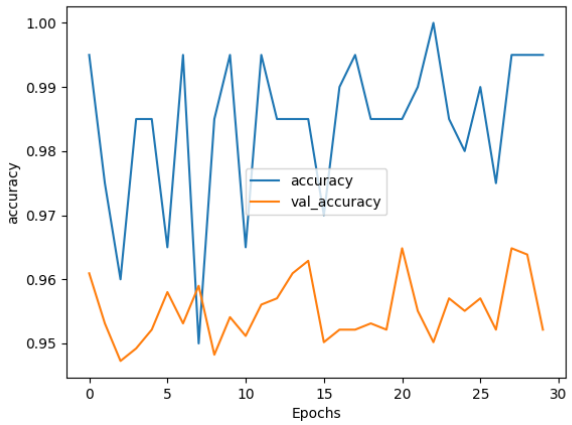
Moving forward to other Neural Networks, I also wanted to compare my RNN model to a Convolutional Neural Network, abbreviated as CNN. CNNs were referred to as even faster to train[4], so I definitely wanted to get some hands-on experience with CNNs.

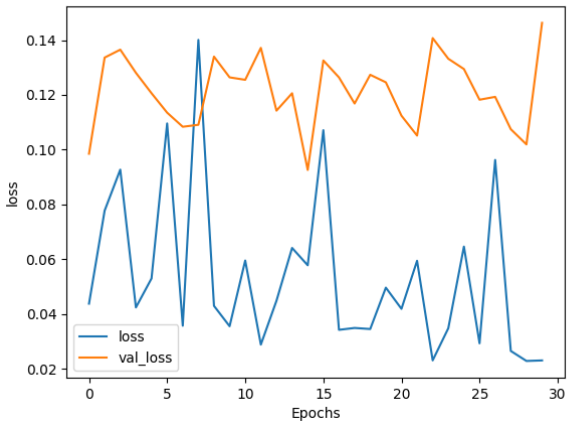
Convolutional Neural Networks (CNNs)

The first training was very promising and took less than 10 minutes on my local machine to compile. The model I had built had a very basic setup with only one convolutional and one pooling layer. I also used googles word2vec[5] embedding model to let the model find better patterns.

Layer (type)	Output Shape	Param #
text_vectorization (TextVectorization)	(1, 64)	0
embedding (Embedding)	(1, 64, 300)	3,000,000
conv1d_1 (Conv1D)	?	0 (unbuilt)
global_max_pooling1d_1 (GlobalMaxPooling1D)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

In training, it was revealed that the model was powerful for a first model, but it was not as advanced as the RNN model I presented. The rules (features) that I wanted to detect in the convolutional layer had a total of 64 and were found in 3-word combinations (kernels). Here are some graphics from 30 epochs of training with partial data exposure of 400 steps per epoch and 1000 validation steps to measure the best.



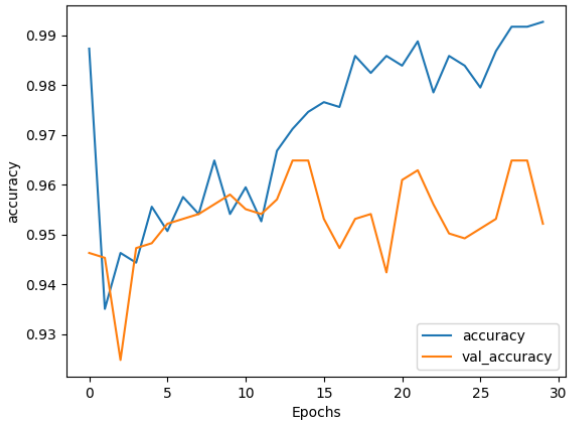


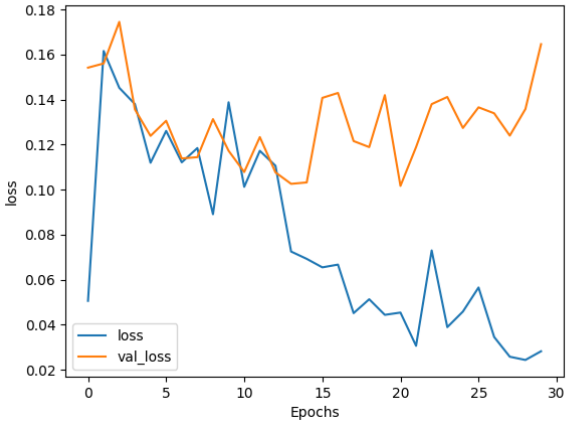
The difference between the accuracy and validation accuracy was only a few percent, which was very nice. However, the gap between the loss and validation loss was huge! I suspected overfitting, so I reduced the features to 32 and ran another round of 30 epochs.

---

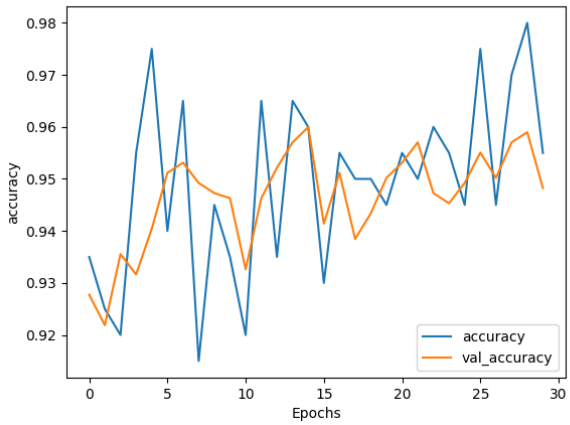
[4] https://towardsdatascience.com/text-classification-rnns-or-cnn-s-98c86a0dd361

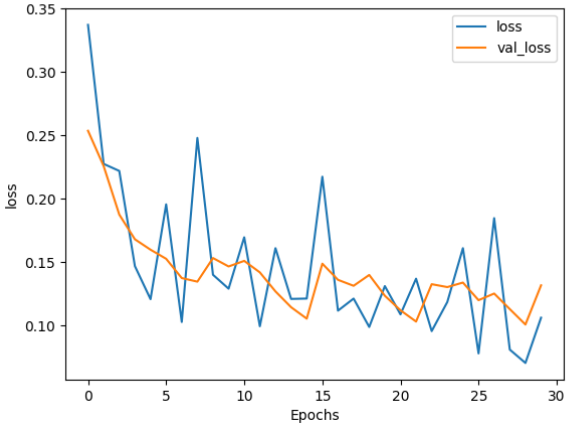
[5] https://code.google.com/archive/p/word2vec/



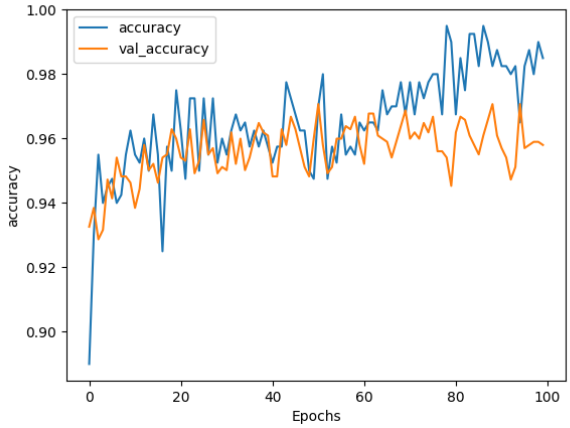


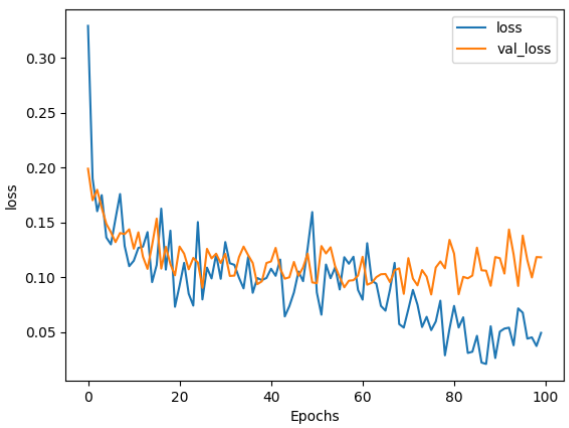
What I detected there was super interesting because with 32 features, the gap between the validation and the training measurements got even bigger. So I reduced the kernel to 3 and hoped that would strengthen the model by recognizing smaller word patterns. The measurements I received were better connected, here the graphs:





Going forward with whatever works, I trained this setup for 100 epochs to see if the loss could drop lower.



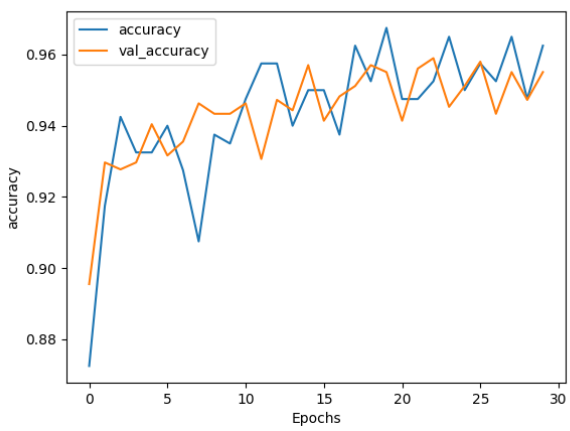


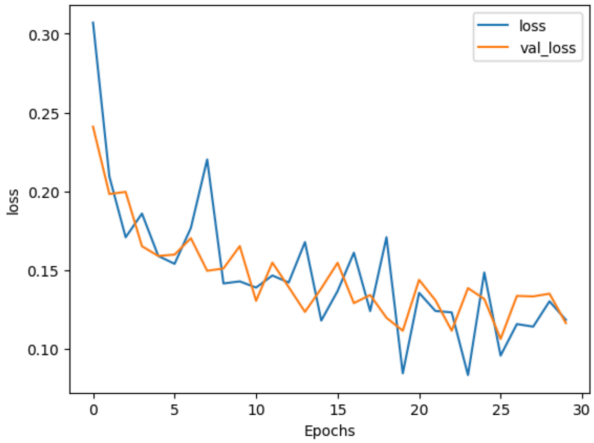
Sadly, we reached the same limit of a 0.1 loss, which is definitely unacceptable because we believe we could achieve better results. After experimenting further by reducing the exposure, but with little success, I added a second layer hoping to uncover more complex structures in the data. Therefore, I included a layer with 64 features following my initial 32-feature layer.

```
model = keras.Sequential([
    vectorize_layer,
    embedding_layer,
    keras.layers.Conv1D(filters=32, kernel_size=5, activation='relu'),
    keras.layers.Conv1D(filters=64, kernel_size=5, activation='relu'),
    keras.layers.GlobalMaxPool1D(),
    keras.layers.Dense(1, activation='sigmoid')
])
```

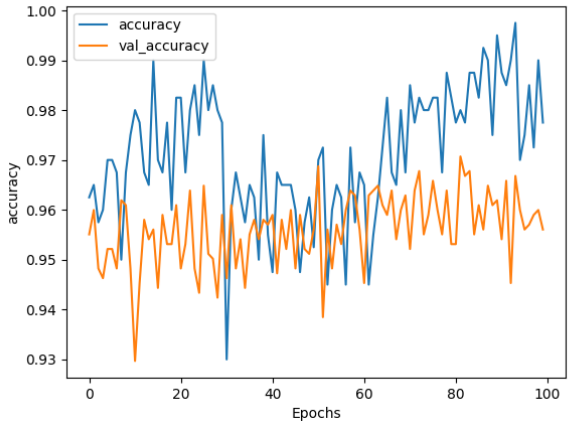
Before showing how this compiled, I wanted to mention how remarkable it is that this model with two convolutional layers has only 58,401 parameters, whereas the RNN model has 714,369 parameters. This is absolutely incredible, considering how far the last model with one convolutional layer came. Additionally, over multiple training rounds, these models showed an increased training speed of 1.5 times.

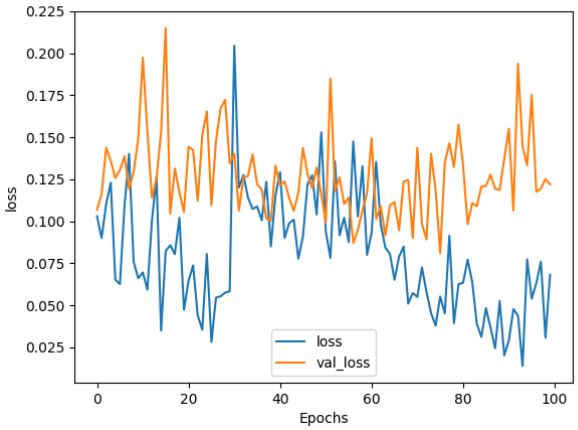
But now, the results:



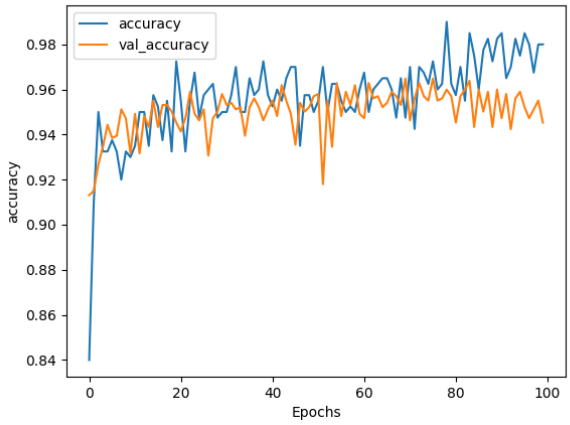


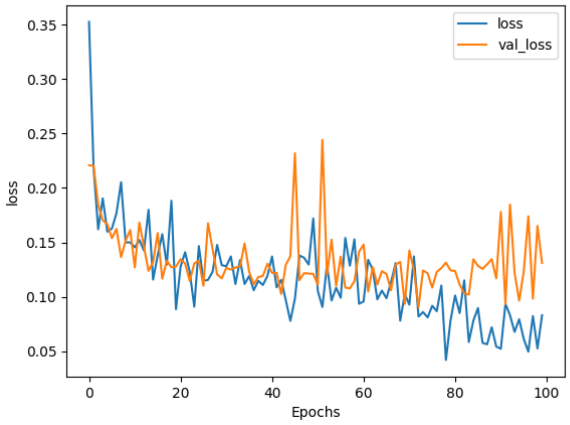
As we see, the training accuracy and the validation accuracy are closely connected, which directly leads me to run the model again for 100 epochs.





This, while looking interesting, sadly showed the same limits, which led me to a last try of adding an extra dense layer at the end to see if the pattern recognition is somehow not able to adapt well enough. An extra layer could help determine if it's possible to use CNNs for this problem because without finding meaningful patterns, the dense layer should also not be able to help the model compile better.





While looking clearer now, we again see the upper limits we saw before, which lets us conclude that moving forward with RNNs is a better approach than CNNs when better results are prioritized. On the other hand, when speed is favored, CNNs are very competitive to RNNs because they can be trained in nearly half the time of an RNN.

Conclusion

As a summary of my work efforts, I want to conclude my learnings and what I believe is worth taking away. Throughout the process of working on these models, I gained valuable insights into the structure of AI models and applied the knowledge about activation functions such as ReLU and sigmoid that we acquired in class. This experience enhanced my understanding of working with neural networks and comparing them. It is worth noting that for complex structures like clickbait detection, RNN models yield superior results on the same database compared to CNNs, although the latter are significantly faster during training. I want to continue working on this in the following months and look forward to finding better models for clickbait detection.