Machine Learning Clickbait

Chris Titus

Table of Contents

Introduction

If you've watched YouTube, odds are you've seen clickbait. Exclamation marks, crazy titles, and strange but intriguing video thumbnails are the hallmarks of clickbait, and are the ludicrous amount of views these videos tend to generate. However, for a given YouTube channel, it may be difficult to know exactly what words to use when writing a title. This works aims to provide a tool to answer this facet of clickbait using a dense neural network. In the process, many hyperparameters are varied in attempt to optimally tune the model for the best results. This project is a continuation and builds upon my previous attempt at this problem last year in CMSC320, so performance will be compared between the two models. Additionally, the model will be "deployed" on another YouTube channel to compare performance.

In my previous attempt at this problem, I employed a binary decision tree algorithm on the term frequency document inverse frequency (TFIDF) which would split on words to classify them as either above or below the median view count of the videos. The topic of those videos were cow hoof trimming videos because some of them had 7 million views and I couldn;t understand how a cow hoof trimming video could be so popular. However, due to the small size of most of those channels, I trained the algorithm on a collection of multiple channels. This was functional, although there were problems that I had hoped to address in the future. This project aims to fix those problems and apply the deeper understanding of machine learning I gained in CMSC422.

The main improvements of this current model are that it uses view count as the "label" instead of above/below median view count and can return a feature importance score for all the words instead of just the words used in the tree.

Processing YouTube Titles

This program is targeting a YouTuber and their channel, so it fundamentally cannot rely on any existing dataset since YouTube channels are constantly making new videos. To address this, YouTube's API was used to dynamically retrieve a list of all videos uploaded by a given channel. This has support for channel information like name, subscriber count, date created, etc. In the old iteration with multiple channels, this was used to account for any relationships between these metrics and video. However, this was changed in the current iteration. Each YouTube channel has its own specific audience, so it doesn't make sense to compare the words that one audience responds to with the words that a different audience responds to. The information used in this model is simply the titles and views, both of which are returned from the video information. All of this data is fetched and stored in a Pandas dataframe for further processing.

In order to transform the titles into a form which can be used for machine learning, the titles are then tokenized, stemmed, and the TFIDF is calculated. Tokenization is the process of parsing the title strings and selecting discarding things like commas and spaces. In both models, only words, '!', and '?' are kept (numbers are not considered). This was done because numbers in the context of video titles tend to carry far less significance than the words. Typically, '!' and '?' are not kept for this process, but clickbait-like titles tend to have many exclamation marks and question marks. I wanted to allow any significance (or insignificance) of these characters to be able to be learned by the model.

Stemming consists of reducing words to their base form. To illustrate this with an example, consider 'Nuke', 'nuked', and 'nukes'. These are each valid tokens from tokenization, and their root meaning is the same, however they are clearly different strings and thus would each have their own feature. We don't want this, so stemming reduces all these words to 'nuke'. What this means is every time 'Nuke', 'nuked', or 'nukes' appear in a title, it is counted as the same word which is what we want (the set of all stemmed tokens is referred to the corpus). Any stop words are removed from the corpus. This allows the TFIDF to be computed in a way that reflects the meaning of the words rather than the verbatim string present in the title.

The TFIDF is exactly what is sounds like and is essentially the number of occurrences of a term in a given document (in this case a title) normalized by a measure of the proportion of documents with the term to the total number of documents. It is used to reduce the influence of words that are used commonly across all of the documents and impart more importance to words which are more rare. The equation used by sklearn is $TF * IDF$ where $TF = \frac{\#\ of\ term\ t\ in\ document\ d}{total\ \#\ of\ terms\ in\ document\ d}$ (term frequency) and $IDF = ln\frac{1+n}{1+df} + 1$ (smooth inverse document frequency) where $n = total\ \#\ of\ documents$ and $df = \#\ of\ documents\ with\ term\ t.$ After this equation has been applied, we are left with a single row of TFIDFs for each term in the corpus for each video title for a given YouTube channel. It's worth noting that the "term" can be an n-gram (a continuous sequence of n items/words). During testing it was seen that n-grams above 1 were less meaningful, and this is to be expected given the small document (in this case title) size which was at most 15 terms in the longest title.

Tune Model

Once the data is collected for the given channel, a model must be trained. This is surprisingly easy with modern machine learning libraries with Python. In an effort to apply lessons from CMSC422, I decided to attempt tuning my model. In my old model, tuning consisted of varying the depth of the decision trees and the impurity measure used, but since the current model is a neural network, there's no applicable comparison to be made. The hyperparameters of the current model that are varied are as follows: optimizer, batch size, train/test ratio, number of epochs, hidden layer size, loss function, and activation function. The results of each with a discussion will be presented below. A disclaimer is that the model will be slightly different each time it is trained, so results will differ slightly than what is shown below. I attempted to get representative results, and when they vary by much it will be discussed. With the exception of the variable being changed, the defaults are as follows (they are also what were deemed optimal from the parametric study):

Table 1: Default Hyperparameters

Hyperparameter Name	Default Value
Optimizer	Adam
Batch Size	128
Train/Test Ratio	0.7
Number of Epochs (if stopping early is enabled, this is the maximum)	100
Hidden Layers	[1024,16]
Loss Function	Mean Squared Error
Activation Function	Rectified Linear Unit (ReLU)
Early Stopping	Enabled
Output Layer Size	1
Input Layer Size (aka corpus size)	~10k (varies with # uploads and by channel)
Max Title Size	15 (varies as above)
Number of Examples (aka number of videos)	~500 (varies as above)
YouTube Channel ID ("uThermal")	UC--TKxqP8xJNymgrLe-thhA

Optimizer

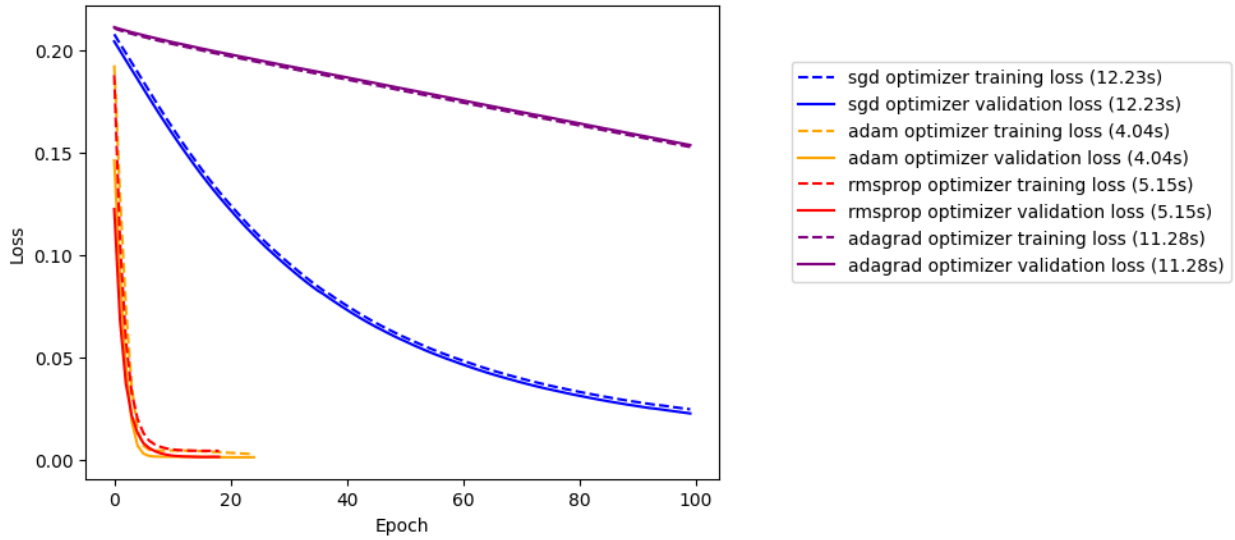

Figure 1: Loss vs Epoch for Various Optimizers

As can been seen from figure 1, adam and rmsprop perform the best, although adam is slightly faster and was thus chosen as the default optimizer. However, it is worth noting that all of these are with the default settings used in Keras. With additional tuning of even just the learning rate, it is likely that the performance of sgd and adagrad can be vastly improved. Unfortunately, this is beyond the scope of this project as it would introduce a disproportionate amount of additional work that is not the focus of this project.

Batch Size

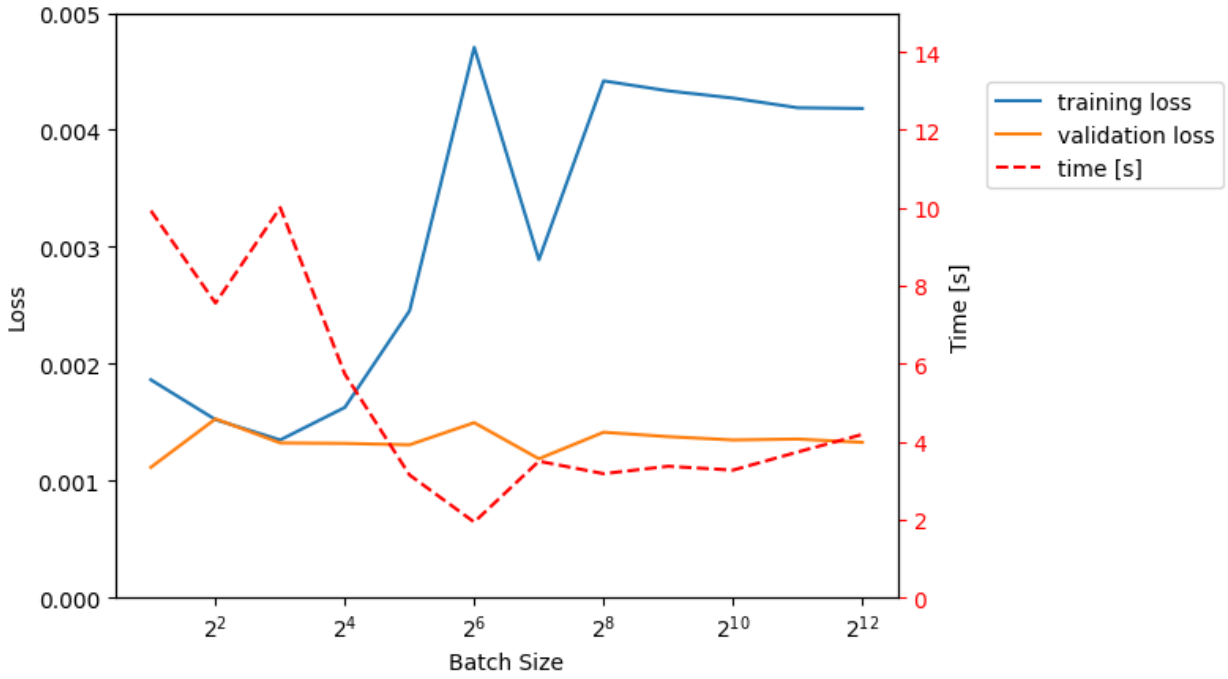

Figure 2: Loss vs Batch Size

Figure 2 shows that batch size does not have as significant an effect as some of the other parameters, but at 128 there is a small improvement. However, time is significantly affected by batch size. I believe this to be a result of the model taking significantly more epochs to converge when trained on smaller batches. For batch sizes above 32, it seems like training converges relatively fast, but for extremely small batch sizes more epochs are required which is to be expected.

Train/Test Ratio

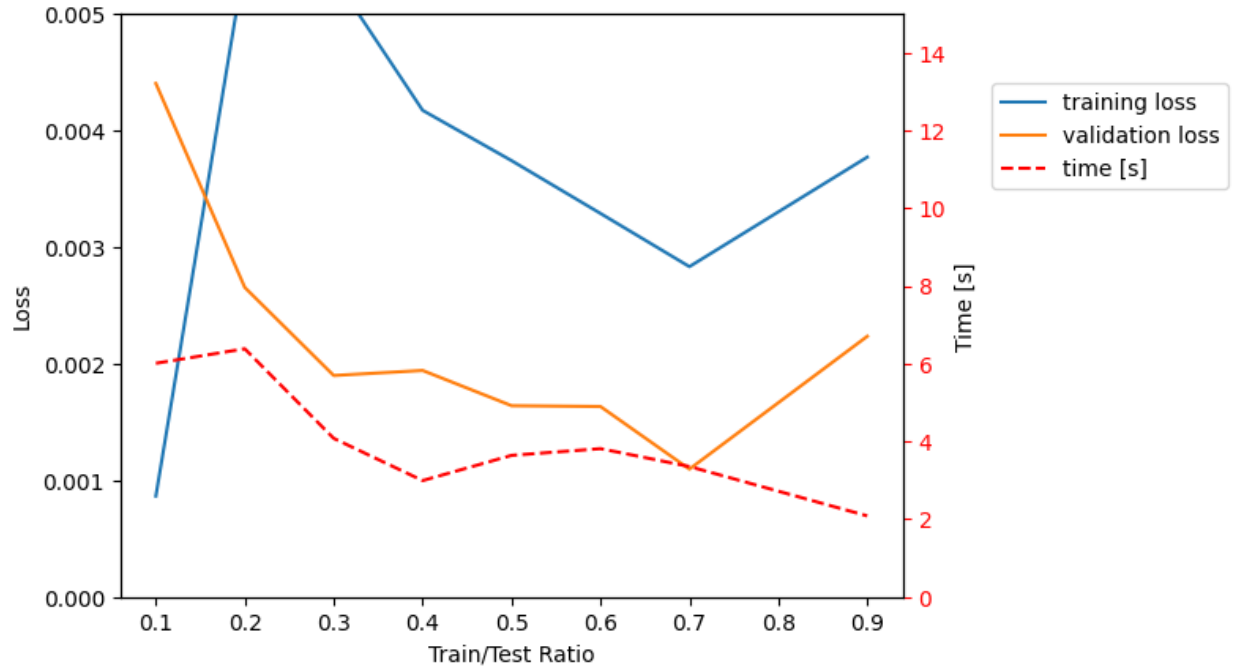

Figure 3: Loss vs Train/Test Split Ratio

As shown in figure 3, train/test split ratios between 0.3 and 0.9 are somewhat acceptable, but ratios from 0.5 to 0.8 are the best with 0.7 being optimal. This is reasonable given the relatively small number of total examples. At extremely small ratios, the training loss is extremely low because the model is able to overfit most of the training data, but obviously this causes quite high validation loss.

Number of Epochs

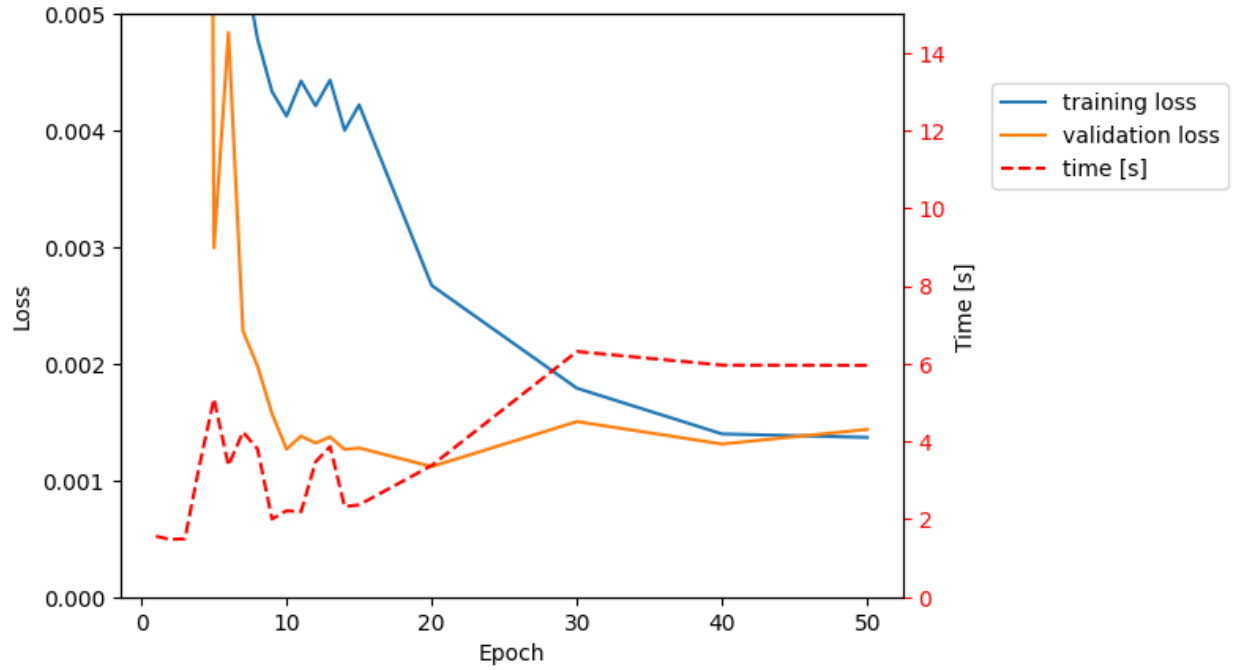

Figure 4: Loss vs Epoch

In figure 4, early stopping is disabled to show the effect of too many and too few epochs. Before 10 epochs, the model is extremely underfit, thus we see both high training loss and high validation loss. Between 10-20 epochs, the model hones in on what appears to be the optimal weights and biases. This is characterized by a moderate training loss but minimum validation loss. At this point, the model has learned enough to generalize well without overfitting the training data. After this point, the model begins to overfit as the training loss decreases significantly while the validation loss somewhat rises.

Hidden Layer Size

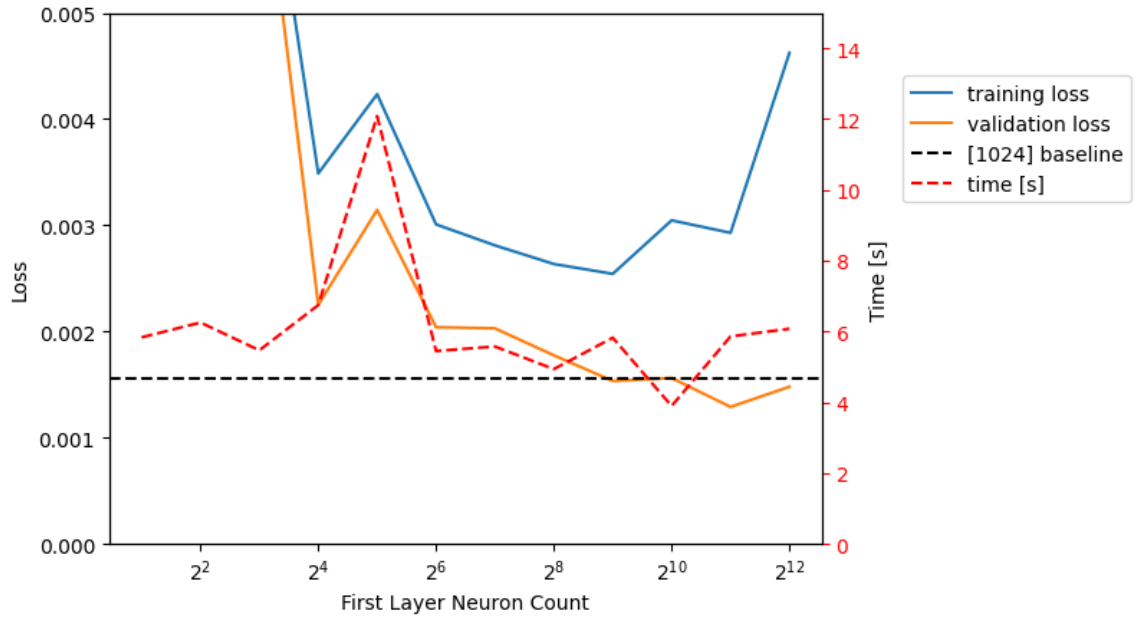

Figure 5.1: Loss vs First Hidden Layer Neuron Count

Figure 5.1 shows that neuron count generally reduces loss. In this specific example, the minimum occurs at 2048, but in other tests the optimal size tends to be between 512 and 2048, so I chose 1024 as the default first layer size. Also worth noting is that each layer was chosen sequentially, so after 1024 was chosen as the first layer it was no longer varied. This is not ideal, since perhaps a [512,X] layer is better than a [1024,X] layer, but this is beyond the scope of this project. More layer sizes were not examined due to the long time running the code would take.

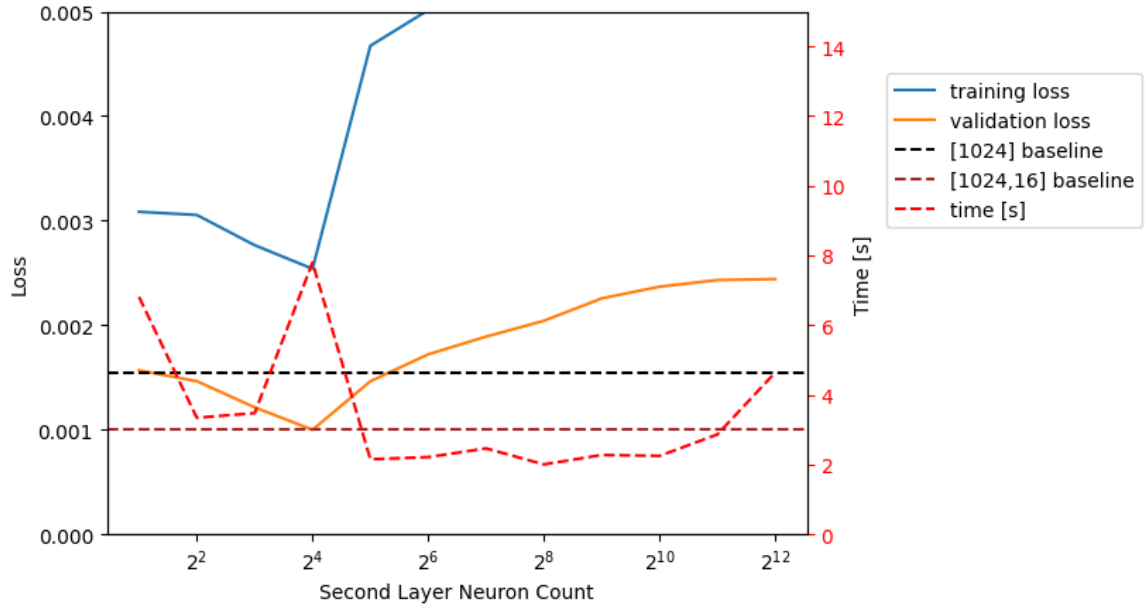

Figure 5.2: Loss vs Second Hidden Layer Neuron Count

Figure 5.2 shows the variation of loss with second layer count assuming a first hidden layer of 1024 neurons. Adding a second layer of between 4 to 32 neurons improves performance relative to a single layer of 1024 neurons with the optimal size being 16. Further testing has had the optimal point between 4-32, so 16 was chosen as the optimal second layer size. There is a strange increase in the time of the 16 neuron case which I can't fully explain. Most likely is that this run took longer to converge, and while this typically results in a worse fit, it appears that this extra time allowed a more optimal solution to be reached. It's also worth noting that the second layer is relatively small compared to the first layer which is relatively small compared to the input layer. This is a typical characteristic of neural networks of this variety because smaller layers sizes reduce the model's ability to overfit data as well as forcing the model to make sure the limited neurons it has are learning the most important features to accurately fit the data.

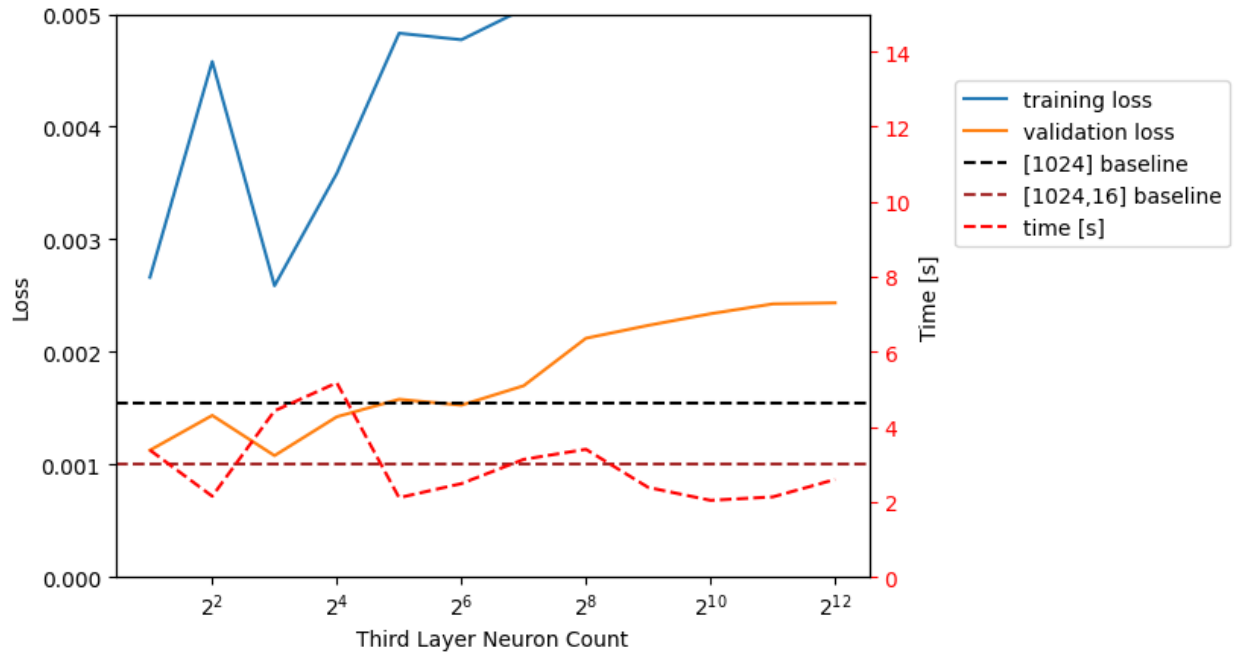

Figure 5.3: Loss vs Third Hidden Layer Neuron Count

Figure 5.3 shows the effect of adding a third layer to the model. As can be seen, none of these perform better than a two layers, so it is fair to say that a third layer overfits this data.

Loss Function

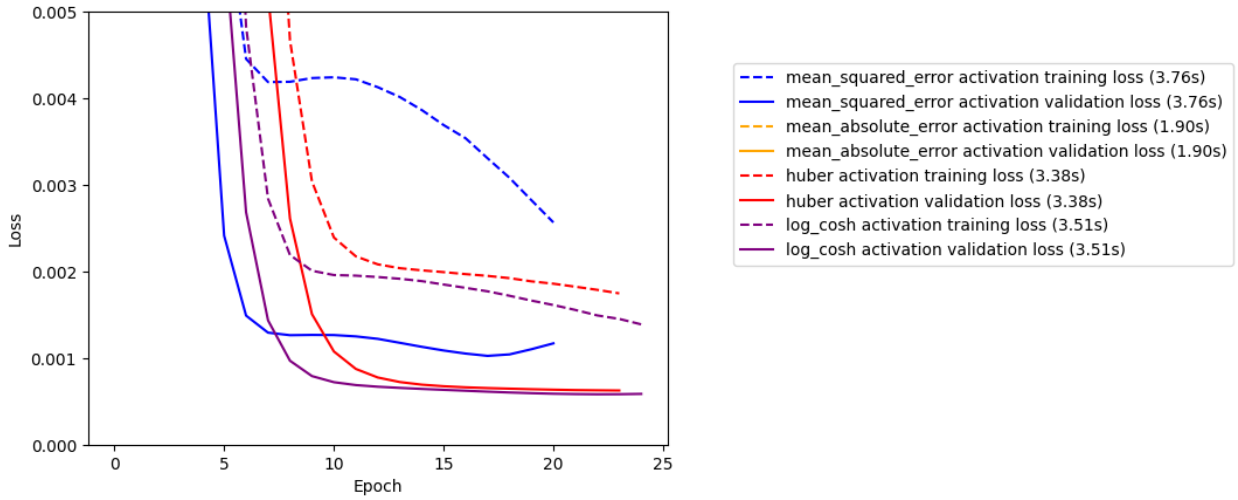

Figure 6: Loss vs Epoch for Various Loss Functions

Figure 6 shows the loss as computed by various loss functions. This is not the ideal comparison, since accuracy would allow for a consistent comparison between different loss functions. However, since this model is essentially performing regression, the loss function is the best we can do, but since each loss function would compute a different loss for the exact same set of predicted values and true values, the absolute value of a given loss function is somewhat meaningless. Instead, we will be comparing the time to convergence. In this metric, mean squared error performs well, but in this particular example too additional time to find a marginally better solution. Typically, mse performs on the same timescale as mae, so mean squared error was chosen to be the default.

Activation Function

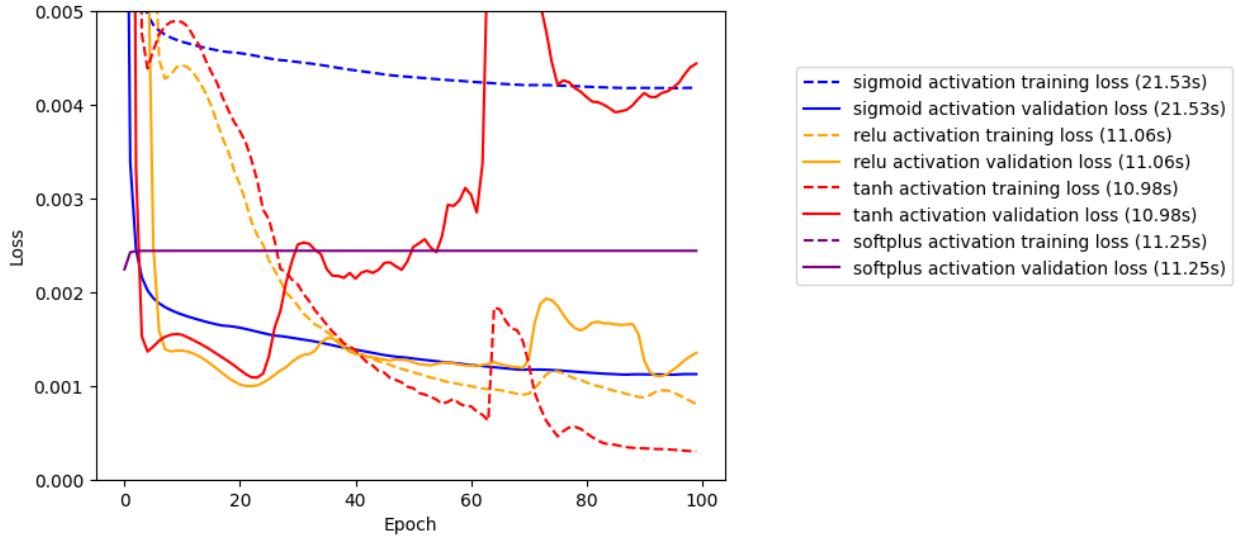

Figure 7: Loss vs Epoch for Various Activation Functions

Figure 7 shows the effect of various activation functions. Early stopping was disabled for this test. As can be seen, ReLU performs the best with a quick convergence and low loss. It also didn't take much time to run which is desirable. Softplus seems to get stuck and does not perform well in this example, but in other examples it would converge the quickest and with the least loss. Perhaps with additional tuning to prevent the instability, softplus would be ideal, but since that is beyond the scope of this project ReLU was chosen.

Results

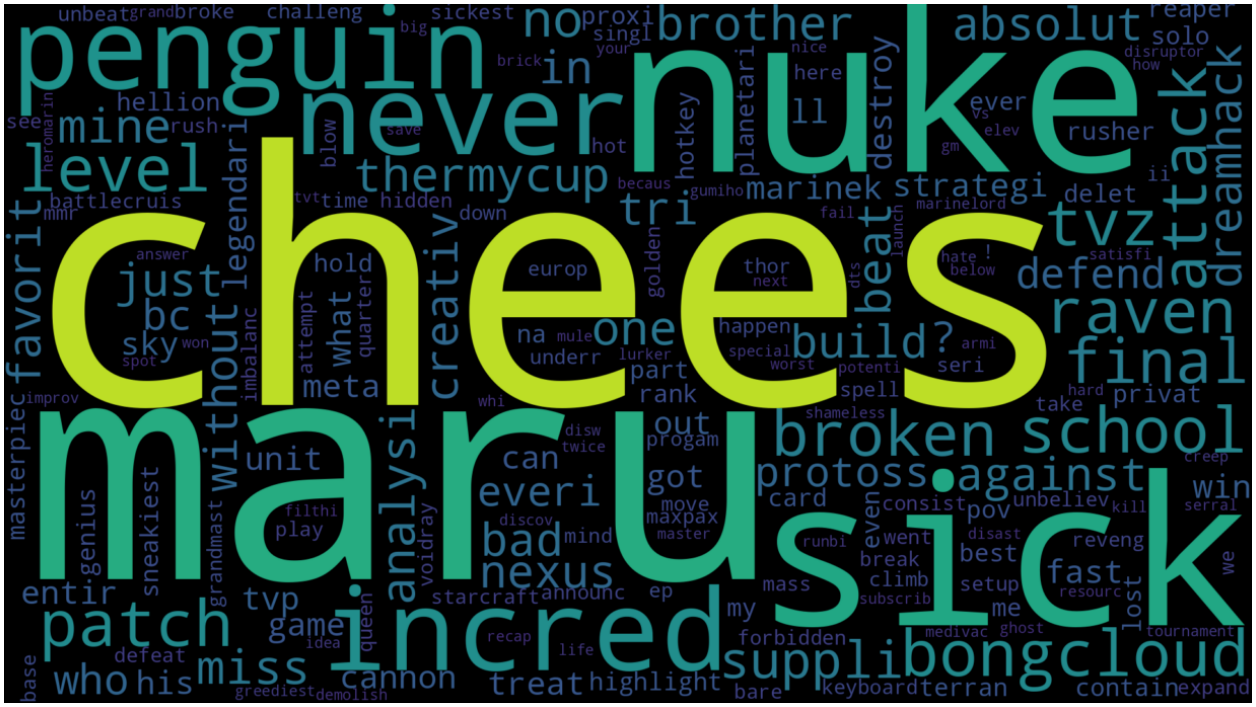

Figure 8: Word Cloud of Top Words For Views

Figure 8 shows the results of running my new model on uThermal's YouTube channel. Feature importance determines the size and color of the words. While many of these words may seem strange to someone unfamiliar with uThermal's starcraft 2 videos, these results are somewhat expected. Chees(e) is the top word, and in the context of starcraft, a cheesy strategy is one which is crazy and unexpected which typically makes for an exciting video. Maru is a top starcraft player, and some of his strategies are covered and played by uThermal, so those types of videos tend to be popular as well. Nukes, sick, and incred(ible), are all words that are indicative of an interesting video. Some strange ones are penguin (bros) and bongcloud which are both hugely popular series with penguin bros and bongcloud being the names of the strategies used in the series. Defend and bad are both unimportant features, and this could be expected because typically you want to watch something where the action is happening on the attack rather than the defense, and why would anyone watch something with bad in the title. Having run this multiple times, the most important work does change often, but many of the top words do remain at the top and many of the bottom words remain at the bottom. Additionally, the placement of most of the words does make sense, so it is reasonable to say that this algorithm does an effective job at determining the highest view generating words!

Comparison

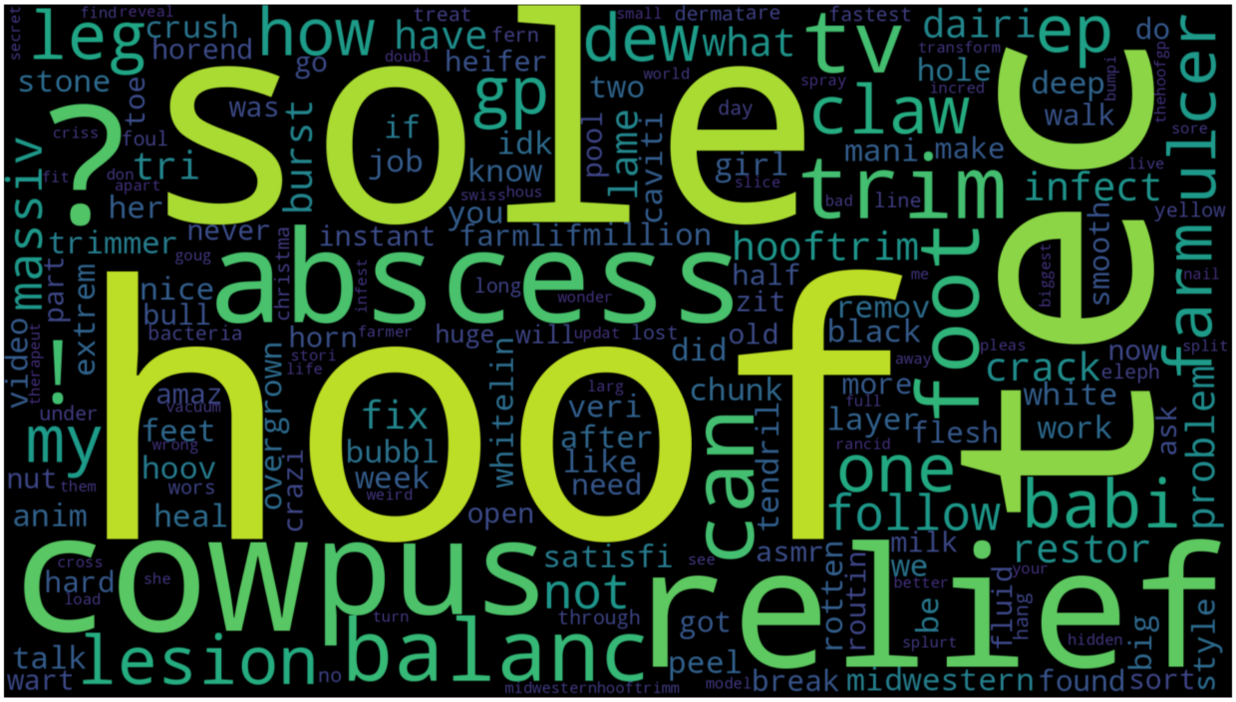

Figure 9: Current Model Top Words on Old Dataset

To further measure performance, the new model was run on the same data as the old model. The old model suggested words like ulcer, cow, foot, and trim were important. Results of the new model are shown in figure 9. There is good agreement between the current model and the old model, but the new model can give the feature importance of all the words which the old one was unable to do.We can also see that there are many words that seem like especially interesting cow hoof trimming videos. Hoof and sole injuries are too be expected from cow hoof trimming channels, and things like pus, lesion, abscess, and relief (for the cow) are all quite clickbait-like and peaks one’s curiosity.

Conclusion/Future Work

Given the comprehensive nature of the hyperparameter parametric study, I am reasonably confident that the chosen neural network configuration is close to optimal within the scope previously mentioned. It returns reasonable results and can be deployed in a real world application on channels it was not tuned to perform well on (ie the cow hoof trimming channels). Importantly, the added capability of working on numeric data and generating a feature importance for all the words instead of just those in the tree has greatly improved the usefulness of the model. However, there is room for further improvement in some areas. Most impactfully, modifying the code to look at different architectures (CNN, random dropout, weight agnostic, and ensemble techniques for example) would be of top interest. Within the current framework, investigating the effect of tuning the loss and activation functions would be useful to have a more fair comparison. Coming up with a way to normalize the different loss function losses would also allow comparison between them. A more comprehensive way to generate stop words (currently I just add any meaningless words that appear in the top words to the stop words list). Finally, un-stemming the words for the word cloud would help with interpretability. Adding the ability to generate whole titles for a given topic would further elevate this model. Despite the lack of these extra features, this model performs better than I would’ve expected at generating a ranking of words that tend to increase view counts on videos and can be quite useful after some interpretation. It also showcases many of the techniques and trends we’ve learned in class.