Classifying Video Labels from Youtube

Roberto Chavez Jr

Advisor: Roberto Manduchi (Computer Vision)

Second Advisor and Main Reader: Ram Akella (Machine Learning)

Thesis submitted for the degree of

Bachelor of Science in Computer Engineering

Computer Engineering Department

University of California, Santa Cruz

Contents

Acknowledgments

I want to acknowledge the reading committee for giving me the opportunity to do my thesis in Deep Learning. Google for distributing their Youtube-8M dataset. Hugo Larochelle for teaching his online Neural Network course. Philipp Schmidt work for open sourcing his yt8m data visualization code from the Kaggle community [46]. Jose Portilla for his online Tensorflow/Keras ml course. “Deep Learning Wizard” for their online PyTorch course. Wandb & Tensorboard for open sourcing their AI tools to analyze performance between algorithms.

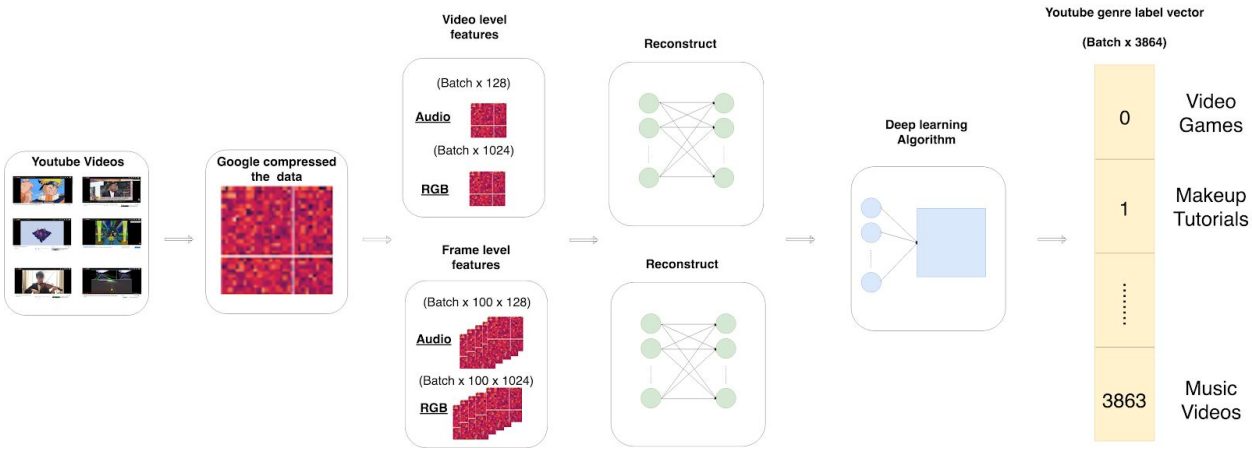


Figure 1: The diagram above is a pipeline of the project. I created four deep learning algorithms, coded 8: 4 in Keras and another in PyTorch. The most efficient algorithm in Figure 2. I used draw.io to generate the diagram.

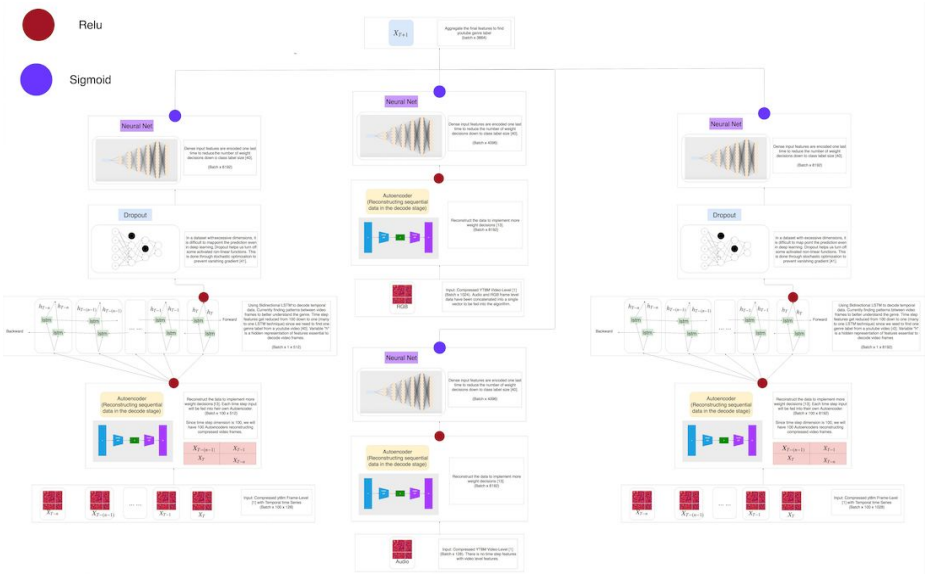


Figure 2: Above is a diagram I made through draw.io of an NN Stream LSTM. My model is too big to display; I'll make sure to make them available on my GitHub.

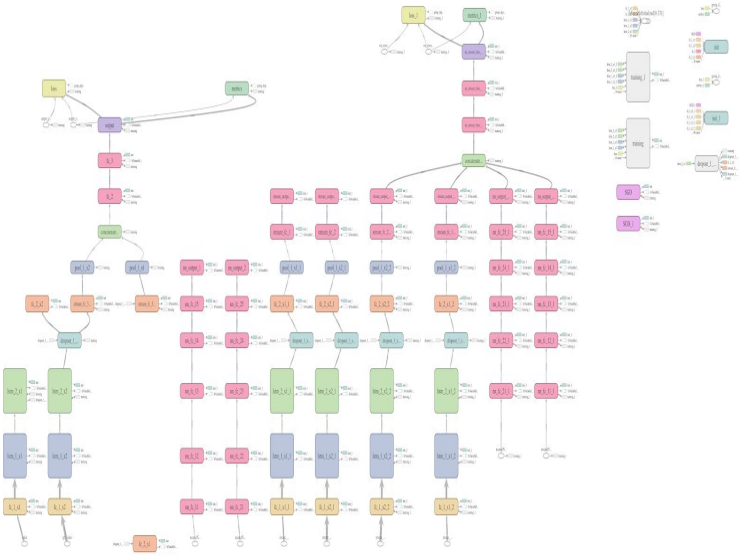


Figure 3: Above is a tensorboard diagram I created for my final algorithm. It contains a combination of Neural Networks to map out dense patterns in spatial content and an LSTM (Long Short term memory) to compute sequential temporal. More info under methods and see the image closer under my Github portfolio.

Abstract

The proposal for my thesis is to develop a classification algorithm assigning genre labels after observing YouTube (YT) videos using the new and improved YT-8M V2 dataset from Google [1]. All have been encoded (compressed) with a Principle Component Analysis (PCA) [1, 7], later had the hidden representation of each video frame extracted before the classification layer. What was once a regular frame of a youtube video, are now compressed with weight parameters to help data scientists to find distinctive pattern between a video to their respective genre [2, 9]. In order to teach a computer to label a genre after watching a YT video, the data needs to be reconstructed from compression using a series of decoding techniques such as generative models. The data will then be encoded again with additional weight parameters to fit the number of extracted features equal to the number of genre labels (output). I coded eight deep learning models: 4 in Keras and the same models in PyTorch to compare not only models but frameworks against each other. From all of my experiments, a temporal neural net |achieves state of the art performance in predicting the correct label. The benchmarks for hardware usage (CPU, GPU, RAM) varied depending on the experiments mentioned later. This algorithm can potentially automate repetitive labor organizing youtube videos with similar content in a recommendation search engine and classify copyright material.

WORK TOOLS USED FOR THE PROJECT

AI FRAMEWORKS

Initially when I proposed this project, I wanted to code with tensorflow for being an open source machine learning framework from Google. After coding with the library for a few months, I self-discovered it's constraints with symbolic programming [36] and horrid syntax. At least with tensorflow, symbolic programming is uncommon in python as it performs computation at the very end of the line instead at the beginning (imperative programming). This tradeoff can lead to frustration debugging since you cannot witness the cause and effect immediately behind each line of code. The upside with symbolic is reusable memory to reference future tasks (generate computational graph). It enables tensorboard to generate a design flow quickly compared to PyTorch.

We also need to visualize the results of our deep learning experiments using Tensorboard to visualize the structure of your algorithm. Wandb helps us create real-time results for training, loss, accuracy, GPU/CPU usage and configurations to design a parallel coordinate (more info on experiment results) [27]. After invoking the keyword wandb.init() onto your training script, an API call is made stored onto an object to run experiments through wandb servers. These two tools will be helpful to observe

the weight decisions and intuition behind each performance. Both deep learning frameworks (PyTorch & Keras) can utilize tensorboard and wandb, all the more reason to have a comparison between the two. Side note: Wandb recently reached out to me after seeing my experiments and wanted me to blog for their company. It's a promising startup to help aid the machine learning community [27].

KERAS

I wanted to code in symbolic programming while avoiding tensorflow syntax style. Keras is effectively a high-level API library built on top of multiple libraries including tensorflow to reduce the number of lines in machine learning. Tensorflow is an open source framework from Google, so it's not surprising a researcher from the same company created keras to ease the length of programming. It's a great deep learning framework for beginners to start with. There are two ways to build a model from keras: sequential and functional. Sequential features allow you to create a set of layers connected back to back but limited to multiple I/O's [38]. For example, if you need to provide one set of features only for audio and another for images, the sequential API won't work. As for functional API, not only you can set up multiple inputs, you can initialize some of them in between layers [38]. One of the most significant downsides I hear about keras is preventing the coder from controlling which variables can be utilized for the GPU or CPU. Two important hardware specs mention later in this report.

PYTORCH

Most AI researchers in the past couple of years have been coding PyTorch for its dynamic imperative flexibility. At the last NeuraIPS conference, most papers awarded used PyTorch. Soumith Chintala, an AI researcher at Facebook, created the GPU-accelerated deep learning library after his horrific experience coding in torch (Lua) to compete in the coco detection challenge [35]. Their network was complicated; it had multiple subsystems it was not always differentiable, resulting in a length of code that was questionable. Lua can optimize their weight predictions only through CPU's which puts a lot of strain on the computer. It cannot handle a multitude of tasks, rendering graphics (cgi-movies, video games) and computing matrix of data processed like a GPU. Soumith also tried coding the coco challenge in tensorflow, but the framework didn't work with his "sensibility" [36]. Now, most deep learning frameworks can utilize GPU for their predictive model. PyTorch gives you the option to decide what functions/variables can either run a CPU or GPU [26]. I attended the PyTorch developer conference, and Google themselves admit PyTorch is an impressive library. Google and Facebook are currently working together to make tensor processing units for PyTorch developers [45] since everyone cannot afford a GPU.

Comparing models and frameworks:

I coded eight deep learning models: 4 in Keras and the same models in PyTorch. I chose to compare not only models but frameworks as well based on research I did on algorithms crashing in production [49]. Tensorflow (Keras) has the largest user base and most traction currently. I want to compare it with PyTorch which is an imperative framework that performs computation as you type it. Tensorflow (Keras) uses symbolic programming: only computing your code at the end of each graph session [38]. Tensorflow is evolving to become more “PyTorch” like with eager execution [51], however that’s still in alpha and didn’t exist at the time my project started. The value of comparing frameworks was also recognized by google engineers as they were working along a similar path in parallel to my work [51]. More information about the difference between each framework and their performance is written in my paper. My experiments reveal surprising results.

FLOYDHUB

Having a computer with no Graphics Processing Unit (GPU) to optimize my algorithms, I invested in a hundred dollars worth of compute credits from a website called Floydhub. Their online GPU credit service allows you to run deep learning models. It was a miserable experience running their site. The platform crashed every time I try to upload 1/3 worth of data from youtube yt8m. I gave up on Floydhub and decided to take the remaining of my money elsewhere.

Paperspace

Paperspace is an online GPU cloud to test/deploy deep learning algorithms. The community for the compute service grew exponentially after Jeremy Howard released his famous fast.ai framework [44]. I met him in person at the PyTorch conference, and he's quite lovely. I then decided to invest in four hundred dollars of compute credits. Their compute service provided an NVIDIA 8gb GPU, 32gb of ram and two terabytes of online storage. To run my code into Paperspace, you'll need to download my latest git commit (GitHub) and upload the content onto your account. Github wouldn't allow me to upload 1.75 TB worth of data which is why I compressed the data into pickle objects [39]. You can run my code onto your local machine, but you will need to download all the imported libraries and have a GPU setup.

Background + RELATED WORK (Previous Research paper)

I will discuss related work to find and or create new methods for the project. Before deep learning, we had support vectors machines (SVM) to classify images & video frames in action recognition with only a success rate of 75% from the Imagenet competition [2]. In 2012 Geoffrey Hinton and his colleagues from the University of Toronto applied a deep neural net in the Imagenet competition, classifying objects on an image with a success rate of 86%[3]. A significant increase compared to SVM and deep

neural nets became the next “moonshot” in the A.I. community. A neural net is a machine learning algorithm inspired by biological neurons in the human brain to analyze data with a non-linear logic structure. Its existence has been around since the 60s with the paper titled “Perceptron”, the potential was lacking due to the lack of data and compute power at the time. The term “deep” is used to add more layers, a stack of weight parameters to increase the level of computer’s comprehension.

The weight parameter for each algorithm aggregates incoming input using the dot product operation, later sent to the output of activated functions, also known as neurons, making decisions from a small portion of features to send a positive or negative signal. If the features are relevant to the youtube genre than the neuron will output a positive message and negative vice versa. In the beginning, before optimizing your parameters, weight parameters are randomly initialized so each neuron will also activate arbitrarily too. Once you start feeding your model with a batch of data, weight parameters begin optimizing a pattern from our input to enable the correct neuron, sending a signal to other neurons would classify the meaning behind a classification. The first layer of neurons handled a small subset of pixels in an image. In between the pipeline, we have hidden layers to aggregate data and deactivate neurons with pixels that are unrelated to the object like the background. The last layer has a higher perception of what the image looks like based on shapes and patterns of lines. Trends are easier to approximate compared to an entire image. Appendix A will have the math explaining more about neural nets and how it ties together for my final algorithm. Set aside Neural nets after observing the data briefly I need to discuss research papers

under unsupervised, feature extractions, and feature aggregation, and possibly identity mapping to help reconstruct and encode the features.

UNSUPERVISE FEATURES

Compressed Data

Since Youtube decoded their data for us [1], we only worry about the high-level features of video frames. The next obstacle is to find related papers to reconstruct the data, later encode it to classify vectors with higher dimensions. The dimensions for the RGB and audio are both smaller than the youtube genre label. One method for reconstructing is super-resolution [10]. A concept to increase the resolution of data like small images while keeping the quality drop to a minimum loss on performance [11]. This technique is often used to enhance satellite images. With raw waveforms, later reconstructed, we have a higher resolution of streaming audio and audio restoration [12]. The only difference now for my project is reconstructing hidden level representations of activated neurons instead of a regular image. More recent methods use autoencoders and interpolation [13, 14] to preserve the minimum loss, creating new quality pixels & audio based on probability from thy neighbor.

Minority Sample

Not only yt8m [1], but most datasets have an imbalance of distributed labels to train the algorithm. In training, a portion of data is used to measure the difference (loss) between the predicted output (algorithm) and expected output (training data output). The smaller

the difference (loss), the more parameters are used to map the pattern between I/O training data. To decrease the damage, we need to send the difference into the optimization process, an attempt to update our parameters (Adam, stochastic gradient) for our next training cycle. For example, given a computer 95 images of a dog and 5 for a cat to train, It'll be difficult for the neural network to tell what a cat looks like if given a new image (test) of a cat. After the algorithm updated its parameter, an attempt to decrease the gap between the predicted output and the expected data; test data measures the final performance of the algorithm without optimizing. Similar to studying, a student repeatedly (epochs) looks at old material (training data), later taking the exam (test data) with no permission from the professor to correct any mistakes (update parameters). What we have is an imbalanced dataset: a biased prediction where the final evaluation of the algorithm will be misleading from the expected result [15]. One solution is to provide more cat pictures. What if we don't have the resources to offer more cat pictures? In unsupervised learning, this is an excellent time to use clustering techniques to mimic input data. In this case, the minority labels, to generalize an imbalance distribution between tags. Most research papers use clustering techniques such as k-means and synthetic minority oversampling (Smote) [16, 17] to help generate more data for the minority labels. Later in the dataset section, I will address the label distribution from yt8m and decide what methods to use.

FEATURE EXTRACTION

Feature extractions are used to uncover input patterns, later help to differentiate between categories (e.g., youtube genre labels). Before deep learning became a trend to extract significant features, there was research in hand-craft representation for action recognition [18, 19]. With the help of local histograms, we can remove frames and motion gradients. The tradeoff was only successful for old images (training data), in other words, overfitting.In machine learning, there are two types of data: training and testing. Optimization (train) is used to update the weight parameters to generalize a pattern from the data given. After training, the test data is used to measure the performance of the model. The weight parameters do not update for testing to see how the algorithm can apply to input in the real world it hasn't seen before. If the algorithm is overfitting from testing, the parameters excessively produce low performance. Similar to a student preparing for a test, their studying (training) by memorizing every text from the book instead of understanding the intuition behind the text [20]. Similar to Google gathering data from yt8m [1], another paper from CVPR was able to extract spatial feature of a frame using a convolutional neural net (CNN) with short term temporal fusion of a pre-trained model [21]. In their case, the model did pooling techniques to reduce the number of features on temporal fusion. CNN models have two techniques in the pipeline: filter and pooling. A filter is used to aggregate pixels by sharing weight decisions, multiplying activated signals with the rest of the neurons in their respective layer to preserve computation compared to a single neural net. The output would

increase the RGB depth channel representing extract features, another dimension. Nearby pixels aggregate similar colors to reduce computation using pooling techniques. For example, a picture of a stop sign is either red or white, only two colors. Using pooling in this example is necessary because we have an excessive amount of red and white pixels for octagon shapes. The shape and colors are typical for the algorithm to decipher compared to looking at the Mona Lisa painting having numerous colors and shapes. Pooling helps eliminate the number of red and white pixels to preserve computation on hardware usage. This outcome captures short term sequential content, making it difficult to compute yt8m for each video encoded is a hundred seconds long [1]. We need to think of a temporal algorithm that can compute long sequential data. Markov chain was among the first to aggregate consecutive time-series data, but the design has limitations to only observe the current state of a time series input to make a conclusion: similar to auto-text for phones to generate the next word. We need an algorithm that is able to observe all one hundred frames, or at the very least close to one hundred, to accurately label a genre from feeding videos into the model. Recent advances with neural networks, specifically in deep learning, have allowed models process sequential data at a larger scale. Last december, colleagues from the University of Toronto attended the NeuralPS 2018 conference presenting their award winning Neural ODE model predicting time series data with remarkable results [48]. Due to their recent discovery, I won't be able to utilize their design in time for my project but work with models currently open source to modify and feasible. Early models of RNN models with neural networks began to show progress observing previous time series

content to influence the current target. However, the early stages of RNN pose a challenge in many applications. They can only leverage a small finate of sequential computation for an input, similar to observing enough words in a sentence to generate a sentiment analysis [53, 54]. Forcing RNN to process longer sequential data pass it's limitations, results in a vanishing gradient [34]. Vanishing gradient crashes the system when we have an excessive set of parameters during training regardless of the size of the training data. The accuracy will have low performance. It's an improvement from markov but we can do better. Now we have models like LSTM and GRU, successors of RNN, that can facilitate longer sequences. They contain additional approximators, also known as activated function, to prevent vanishing gradient. The second method would involve taking the feature distribution of videos through maximum pooling. Most common pooling techniques are vector of locally aggregated descriptors (VLAD), fisher vector or bag-of-words [23, 24, 25]. Since I displayed the distribution for each RGB video label, it's an option worth considering. Poolings are not trainable but they are an operation that can reduce training time by dividing the input map, downsampling per frame on a youtube video. More issues I hope to address under the method section.

Identity Mapping

If the data from yt8m becomes too complicated for standard deep learning models to train, I will need to find an alternative method to address my issue. Adding more parameters into a deep learning models initially sounds promising since the term "deep" came from adding more parameters into a neural network: more expressiveness, better

generalization. However, most cases adding an excessive amount of parameters can cause two things: overfitting and vanishing gradient. Microsoft tried to address these issues in the Imagenet competition by creating a new deep learning model called Resnet [35]. The architecture used a technique called Identity mapping, enabling you to skip parameters at random times through optimization. During training, it's essential to deal with fewer settings to prevent vanishing gradient while adding more to generalize. Microsoft uses Resnet to encode spatial data, but why not apply the same technology for continuous data? More info under the methods section.

ANALYZING THE DATASET (Philipp Schmidt)

There were two crucial tasks Google researchers [1] developed with the video dataset on a large scale.

1. Google researchers are scaling time-consuming videos to annotate manual images. Resolving this issue identified relevant knowledge graph topics for all public youtube videos. Any URL site with more than 1000 views, seeking diverse vocabulary of entities, and 24 top-level vertical categories popular on youtube were considered relevant knowledge.
2. Scaling videos are computationally expensive to process. Initially dealing with a petabyte of video storage and dozens of CPU-Layers process may seem impractical for students. Google researchers successfully pre-processed the videos and extracted frame-level features using deep learning in image recognition and PCA dimensionality. Elements were removed 1 frame per

second from 1.9 billion videos and were able to compressed petabyte content down to 1.5 TB.

In the upcoming sub-sections (**data processing, distribution of classes in the training set, probability of label occurrence**), I will discuss Philipp Schmidt work for open sourcing his yt8m data visualization code from the Kaggle community [46] to summarize the data visually. I'm summarizing Phillip's kernel not for my benefit but help readers understand the data visually and to better understand the reason behind my proposed machine learning **methods** in the next section. I saved all of the data graphs from Philipp onto pickle objects [28], processing all of the data would take weeks, maybe up to a month. I did perform feature engineering, making the data clean to feed into my algorithms with unbiased mathematical patterns.

Data Processing

Traditionally machine learning algorithms don't require a mass amount of content. yt8m contains 1.5 terabytes of content; it'll take a while to train and process all of the data onto a program. Which is why I need "deep learning" to help scale a large amount of data. We need a respective CPU/GPU (Paperspace: Nvidia 1080) and an algorithm that can scale large data. To process all of the content, we will need Tensorflows' API framework "tf.python_io.tf_record_iterator" to iterate each file and organize the I/O [1, 46]. Processing 1/4 of the dataset became time-consuming since it took weeks; close to a month. Afterward I compress a portion of the content into "pickle objects" for a quick demo since GitHub has a limitation on storage [28]. If you are interested in feeding in

more data to train on, the code below, available on my Github (Algorithms.ipynb), shows a snippet how to process video labels.

```
# Video: Trained labels
file_Number = 0
vid_ids_train = []
mean_rgb_train = []
mean_audio_train = []
video_labels_train = []
for file in video_files_train:
    print("file_Number: ", file_Number)
    file_Number += 1
    for example in tf.python_io.tf_record_iterator(file):
        tf_example = tf.train.Example.FromString(example)

vid_ids_train.append(tf_example.features.feature['id'].bytes_list.value[0].decode(encoding='UTF-8'))
        video_labels_train.append(tf_example.features.feature['labels'].int64_list.value)
        mean_rgb_train.append(tf_example.features.feature['mean_rgb'].float_list.value)
        mean_audio_train.append(tf_example.features.feature['mean_audio'].float_list.value)
mean_rgb_train = array(mean_rgb_train)
mean_audio_train = array(mean_audio_train)
```

Figure 4: Above is a list of code to parse data from google's database [1, 46] encoded with tensorflow to compute data with on a GPU. With the help of the kaggle community, we need to initialize empty data structures to hold the train/test data after downloading it from youtube.

What we have above are two for loops: first loop process one file at a time while the second looks at the number of youtube videos inside the file. Each youtube video has a set of features ("id," "labels," "mean_rgb," "mean_audio"). Audio, RGB and labels are stored in array pickle objects to be accessed immediately to be cleaned; perform feature engineering or analyze the label distribution between youtube genre labels to see what set of data is not unbiased with its distribution.

DISTRIBUTION OF CLASSES IN THE TRAINING SET

Looking at the distribution between each label is essential to know what techniques in feature engineering are necessary to prevent unbiased prediction. From my experience, oversampling (provide more data), manipulating features, or generative models can help attain a balance data set and utilize simpler models with higher accuracy. From a small sample size of 6 labels out of 3846, My computer couldn't process all 3846, they are graphed below on **figure 5** to measure common features between each genre. The x-axis represents the group size of labels having similar content to another genre (i.e. racing genre may look similar to rail-transport). The y-axis is a measurement of how common the groupings are between labels. For example, a toy genre may look similar to an airplane video if the airplane was actually a toy. Based on a small sample of the data, cosmetics, with a sample size approximately 20% and a group size of 2 have a higher placement than airplane videos with ~12% and a group size of 6. Airplane videos content may confuse the model for not being unique and get confused classifying a youtube video unrelated to airplanes.

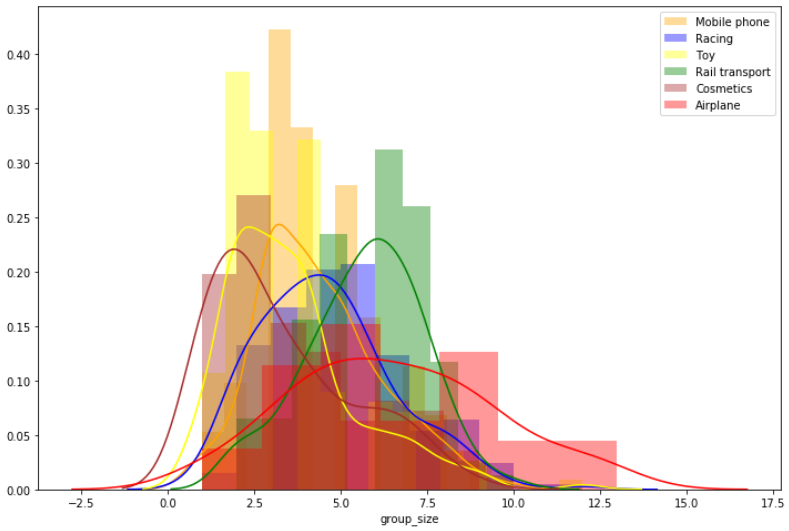


Figure 5: Above we have ourselves a label count distribution [46]. On average each genre label has been grouped by 3-5 types of videos. Work was contributed with the help of Philipp Schmidt [46].

PROBABILITY OF LABEL OCCURRENCE

Below is a similarity matrix coded of the estimated probability of label occurrence, given another label from sample data. We have a for loop coded: the first loop recording the sample data, second calculates the similarity measurement between tags.

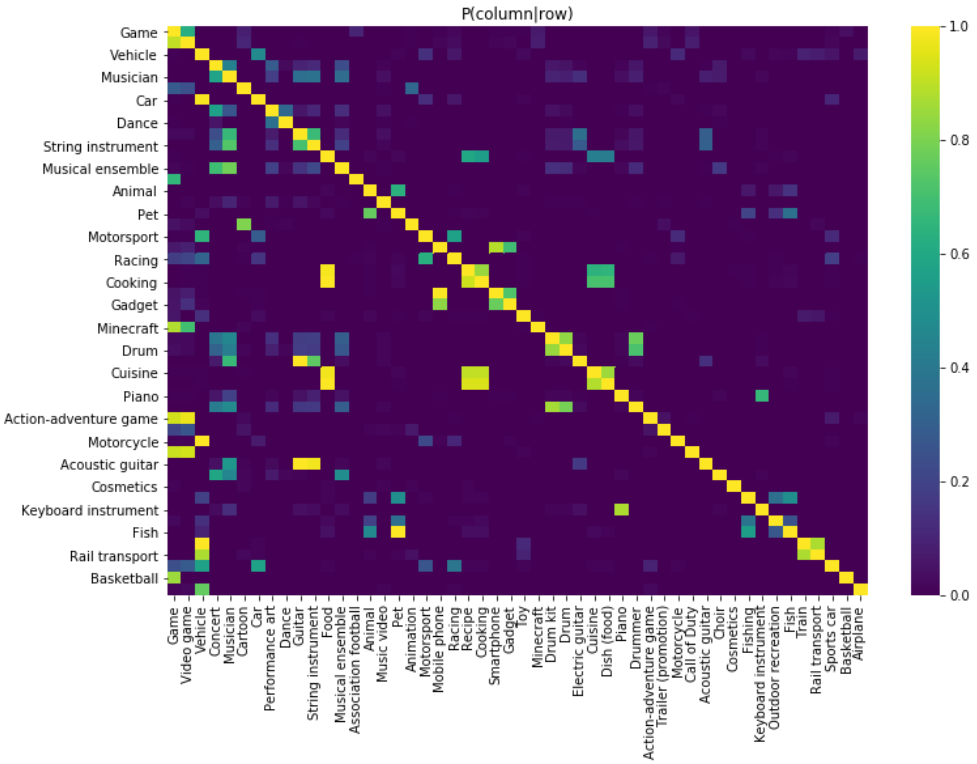


Figure 6: Above we have a similarity matrix representing the similarities between the labels since youtube videos have more than one genre [46]. An average video has three genre labels, but the data will only have one name-tag each to make sure the least favorite videos to make unbias prediction. All can be visualized from the jupyter notebook code. Work was contributed with the help of Philipp Schmidt [46].

The label occurrence from the samples on **Figure 6** is not always identical between P(A|B) and P(B|A). If you look at the label "Games" at row 2 P(Games|Every Column),

we only have one high probability which is P(Games|Games). Whereas when we look at Games on column 2 P(Every row | Games), we have a high probability for "Call of Duty: Ghost," "String Instruments," "The Sims," "Football" and lastly "Games." Another thing about the heatmap that caught my attention, why P(Car|Cosmetics) is so high and P(Car|Vehicle) is low? These two occurrences would make more sense if their results would be the opposite for a car looks like a vehicle more than cosmetics. Either there is a bug mapping the labels to their actual names or this happens due to small sampling, since we don't use a lot of videos to estimate these probabilities. Overall the heatmap does show promising label occurrence, in the top 100 common videos uploaded on youtube, to feed into a machine learning model. If the performance for my algorithms is relatively low, I would need to compare more distributions on the least favorite videos on youtube (real estate) and synthesize more data for them. Now that we have analyzed the data sample it's essential to observe the video and audio features and how can I create my model. At times a model can predict a label from a video similar to a genre we attended to classify. For example, if I fed a soccer game video into the model, the AI can mistake it for a basketball game due to the lack of features or histogram for an object shaped like a ball. Only way an AI can tell the distinction between these two is to provide an even distribution between each sport alongside every genre in the database.

VIDEO LEVEL RGB FEATURES

If you wish to make a quick machine learning algorithm to predict a genre, video-level features should be your first place to analyze. There are no continuous

time-series data in video level data, an additional set of matrix-dimension to compute unlike frame-level features. Spatial elements have already been compressed unlike temporal data so you will expect performance differ when optimizing algorithms for two different set of features.

On **Figure 7**, philipp randomize which RGB labels should be graphed below to explore what needs to be regularize [46]? What this tells us for RGB features; every data set with video label doesn't need to undergo regularization since the class distribution looks balanced compared to frame level features. Regularization reduces the number of parameters to prevent overfitting: an excessive amount of weights to only memorize content from training instead of generalizing a mapping pattern to predict test data. With neural nets, we use dropout for regularization. **Figure 7** we see the distribution between labels in video level features: it turns out the genre names are equally balanced. All we need from the graph is to remember the grouping values between -0.2 to 0.2. Every value outside the range is close to zero for every distribution. The data, at least for RGB features, is unbiased and will generalize the prediction for our model.

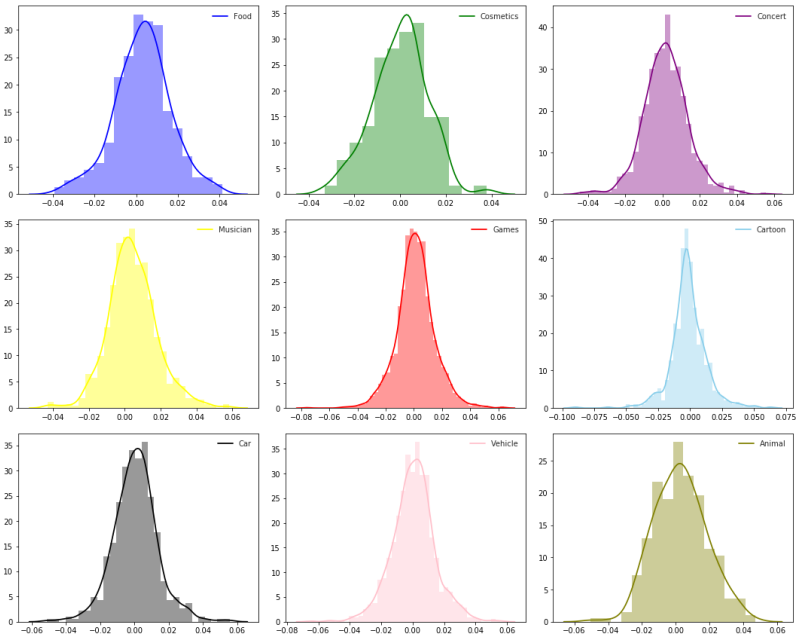


Figure 7: Above we have an equal distribution between each label since we're only training with one genre per video. I randomize which genre would appear above from 3864 videos, giving me an unbiased perspective. Work was contributed with the help of Philipp Schmidt [46].

FRAME LEVEL FEATURES

We examine the audio frame level features of a youtube video [1,6]. The frame features extracted a sample each through a series of video frames [1, 6]. We are dealing with continuous data frames containing audio spectrum [1, 46], it's important to know the higher distribution to set a regulation and equalize the number of data per frame.

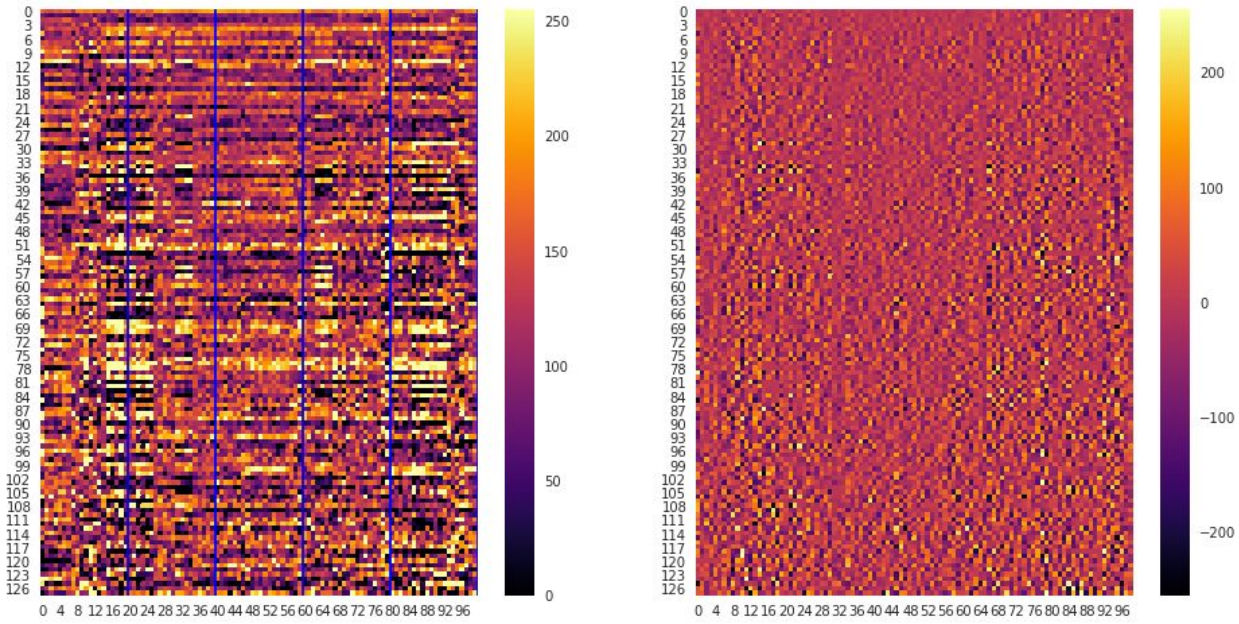


Figure 8: Above we have a 2-dimensional array of a frame containing features from a 100-second youtube video. For each row we have a spectrogram display frequency while the columns are the timestep. Work was contributed with the help of Philipp Schmidt [46].

On **Figure 8**, we have a two-dimensional spectrogram of a video. Each row displaying frequency whereas each column is a timestep (100 seconds). The color represents the magnitude of the audio frequency. It seems like each video has a high pitch frequency magnitude, most common samples colored yellow, in the beginning and end to a youtube video at time step 20 and 88. This high pitch can be the primary source of information we can config when coding a machine learning algorithm; one that is able to approximate temporal models. Approximate sequential data both in the beginning and end. Now that we have a good understanding of the dataset, we can make theories which machine learning algorithms can help classify youtube labels as accurate as possible.

Methods

In the upcoming sub-sections (**Feature Engineering, Integration, Algorithms**), I will discuss the methods used to find an efficient pattern knowing the distinction between data labels (genre).

Feature Engineering

I reconstructed the data using an autoencoder with additional weight parameters making sure the compressed input content could fit a large genre label. The output vector maps the index for each genre (E.g., Games, Art & Entertainment, etc.) to their respected youtube video. Once the data has been reconstructed with additional parameters to match the same dimension as the output, we can then choose a series of deep learning algorithms, later covered in the sub-section (**Algorithms**).

Integration

We are dealing with compressed data containing two important features we need to aggregate: video-level and frame-level content. We need to integrate three different algorithms into the pipeline:

1. Reconstruct the compressed content [1] with initialized parameters. The number of features is smaller than the number of class labels (output genre labels) which is why I had to reconstruct the data. Add more dimensions into the input data. More details available in the paper under the Feature Engineering section. The

output of the reconstructed features section. The output of the reconstructed features in step (1) are separately sent into their respective models in (2) and (3)

2. (2) Compute sequential data (frame-level) for each video gathered from yt8m. Each video is at least 100 second long to be utilized within the dataset. We can use temporal models (Recurrent Neural Net).
3. (3) Instead of 100 second per video, Google created video-level features extracting a task-independent fixed-length vector per frame. In other words video-level features have one less dimension from compressing frame-level sequential content. We can train this data using classifiers like logistic regression or any non-linear design.

Both models (2) and (3) models contain a softmax approximator model to determine the label of a genre from a youtube video. You can develop an algorithm (section below) with both models separately or combined (concatenate function for Keras & PyTorch).

Algorithms

Below are four algorithms I coded to find the model with the highest probability to accurately label a youtube genre per video. Later I compare each framework (PyTorch & Keras) with the same models to see which one has the most efficient for research and or production.

1. **Fully Connected Net**: Aggregating compressed spatial features from youtube videos

2. **Bidirectional LSTM**: we are using a Bidirectional LSTM to aggregate sequential content of video frames. Each youtube video has been cut down to 100 seconds of frames. Any video less than 100 seconds is not part of the dataset to balance the distribution. For example, if there was a 10 second youtube video of nascar-racing inside the dataset while another video with 100 seconds of a video game of cars, the algorithm would likely have a bias prediction labeling a nascar-race as a video game
3. **Stream-LSTM**: The model is similar to Bidirectional except we have the approximators for audio and rgb separately sent into their own fully connected nets. The approximators are later combined into a concatenate function to find the average. For example if the approximator for one genre label in an rgb vector was 79% while the audio was 99%, concatenating both would be 89%
4. **Fully Connected Net (Video-level) concatenated with a Stream-LSTM (frame-level)**. It's a combination of algorithm (1) & (3) softmax approximation using concatenation.

Training a model requires data to help the model update its parameters similar to studying for an exam. This process is repeated multiple times until there are no more training batches and epoch iterations. After explaining each model in this thesis, I will give a step by step solution translating my algorithms into Keras & PyTorch.

Autoencoder (Decoder-Net)

The data for spatial feature is compressed, and the original youtube videos (data) are not open sourced, I think the second half of an autoencoder, “decoder-net”, is an ideal choice to reconstruct the data [13]. A traditional autoencoder mimics the original input after encoding (dimensionality-reduction) with a decoder (generative); however we do not have the original videos but only yt8m. What I’m proposing is utilizing the second half of an autoencoder to provide additional weight parameters multiplied with the input to increase the dimension via generated features that are optimized. Once the data has been reconstructed with additional parameters to match the same dimension as the output, we can then choose a series of deep learning algorithms to decide how to encode again but with additional generated parameters. Sending yt8m onto a deep learning model directly will only compress the data even more. Remember Google encoded the youtube videos with a PCA to decrease the size of the data from 0.5 petabytes down to 1.5 terabytes; enabling data scientists to download the data.

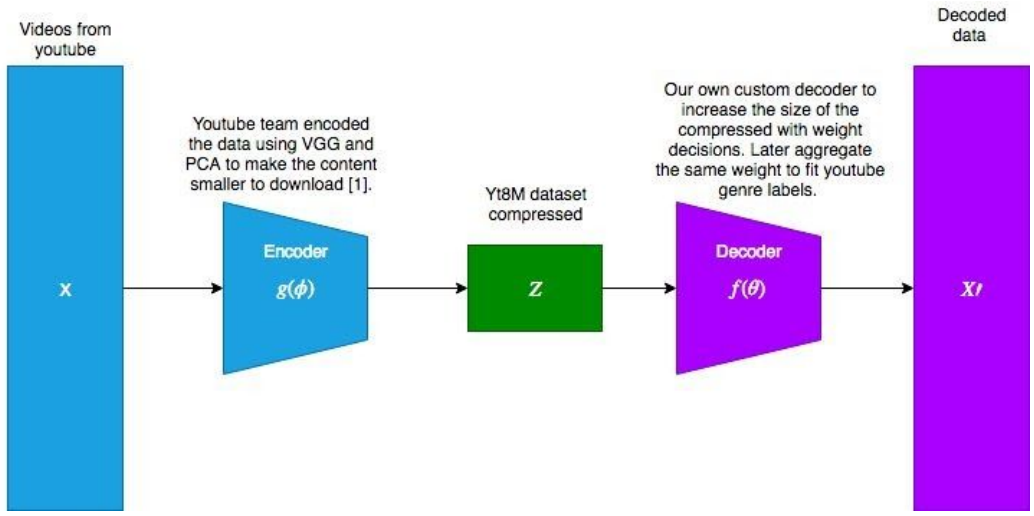


Figure 9: Above we have an autoencoder [13] which is in the form of neural nets reconstructing the data. I made the drawings on draw.io. I plan on utilizing the second half of the model to support generated dimensions for having a smaller input size than output.

For the autoencoder we're given data to reconstruct compressed latent space data, in this case yt8m (input), to fit the class label size. The graph above indicates our given data "Z" instead of "X" because youtube already compressed the videos to shorten the memory size for download. To have more parameters, we aggregate our input using the dot product with initialized weight parameters. Making the compressed videos trainable.

Z: YT8M Compressed Dataset

Ø: Aggregated weight parameters

Reconstructing our new input: $f(Ø) = Z * Ø$

Fully Connected Net

The model has video level rgb and audio features fed into a one-vs-all binary logistic regression. Binary logistic is another word for nonlinear function mapping data with a large number of features like encoding objects in an image. Each label trained on this model uses video-level features with no frames to measure the performance. These weights multiplied with youtube frames, later output a probability number between 0-1 using activated functions. Based on recent studies, relu is the most optimal function to prevent vanishing gradient [30]. Each set of inputs and weights fed into the model are multiplied then sent into a rectifier function. If the output of the function is above zero, we will have a linear function: derive into a flat slope at one to prevent vanishing gradient. If the output of the rectifier zero or below, we have a slope of zero and the

weights associated with the function will not be able to contribute and teach the AI to label (vanishing gradient). Once I'm done creating hidden layers for the weights, I concatenate both audio and RGB into one vector and have them fed into one more encoder to fit the size of the youtube genre vector. During training, the model uses a binary cross entropy penalty with a sigmoid function to calculate the lost.

Algorithm 1: Using a neural network to aggregate compressed spatial features from a youtube video.

Input :
1 W: Weights for the perspective layers
2 Dimensions for video-level x-inputs: (Batch x Features)
3 x_audio: Compressed PCA video-level content of audio (64 x 128)
4 x_rgb: Compressed PCA video-level content of spatial footage (64 x 1024)
5 Y_{data}: genre labels for records in x data
Output: Y_{T+1} Classified genre label of a youtube video
6 for epoch in range(epochs):
7 for i, (x_audio, x_rgb, Y_{data}) in number_of_batch_examples:
8 concat = contatenate((x_audio, x_rgb), dim=1)
9 Y_{T+1} = Neural Net (concat)
10 $Y_{T+1} \leftarrow$ Rectifier Activated Function()
11 $Y_{T+1} \leftarrow$ Decoder Net ((x_audio, x_rgb), W)
12 $Y_{T+1} \leftarrow$ Feedforward (σ(x_audio, x_rgb), W)
13 $Y_{T+1} \leftarrow \sigma(Y_{T+1})$
14 Loss = CrossEntropyLoss(Y_{T+1}, Y_{data})
15 Append Loss into an array of losses (graph out report)
16 Adam Optimizer (Y_{T+1})

Figure 10: Above is a latex image I coded representing a pseudo of my first algorithm. The fully connected net is located within line 9 to 13 being trained inside a for loop of batch data.

CODED IN KERAS

Using Keras' dense API function (linear), we initiate weight decisions to do element-wise multiplication, aggregating the input. Inside the dense keras function we have an argument called "activation," enabling you to pass the string "relu" and use the rectifier function. Once I finish coding the hidden dense layers, I concatenate two hidden layers (RGB & audio) into one using keras.layers. Invoke sigmoid as an activation for the final dense layer, also known as the output, and train the model using the appropriate model class.

CODED IN PYTORCH

Since PyTorch is imperative programming, I can code my model using object-oriented programming. We create a class to support the model architecture and a forward function to pass in input data, return output from the model. I initialize my weights, layers and activate functions operations inside a python class called "Neural_Net." The benefit of creating a model class in PyTorch save private variables and are wrapped through nn.Sequential, a class from PyTorch to add modules: including layers or activated functions. Modules added in the order they were coded are how they lineup in neural net architecture; this is similar to keras dense layout except we can dynamically control the parameters and GPU usage. All of these variables later reference forward functions, manipulating input data to output something meaningful. The output vector contains either a one or zero: one represents the classified label, and the other should

be zero for having the lowest approximation. Architecture is finished, and I used “ nn.CrossEntropyLoss”, a criterion in PyTorch that measures the Binary Cross Entropy.

Multi-Bidirectional LSTM

This model requires two subsections: One explaining the history behind RNN & LSTM and the benefits having multiple recurrent models into a pipeline.

Algorithm 2: We are using a Bidirectional LSTM to aggregate sequential content of video frames. Each youtube video has been cut down to 100 seconds of frames. Any youtube less than 100 seconds is not part of the dataset to have a more balanced distribution to train. From the second layer of LSTM, we only need the last hidden state, theoretically speaking, it should represent a vector having an understanding of every frame in the youtube video. Even though the algorithm has the name "long," recent breakthroughs in sequential models in 2018 called transformers [47] has shown LSTM "alone" has flaws with no neural net inbetwen to approximate longer content (Stream LSTM).

Input :
1 (b): Memory blocks bias
2 (W): Weights for the perspective layers
3 (STM): Short term memory initialize random parameters
4 (LTM): Long term memory initialize random parameters
5 Dimensions for frame-level x-inputs: (Batch x 100-Frames x Features)
6 x_audio: Compressed PCA frame-level content of audio (64 x 100 x 128)
7 x_rgb: Compressed PCA frame-level content of spatial footage (64 x 100 x 1024)
8 Y_{data}: genre labels for records in x data
Output: Y_{T+1} Classified genre label of a youtube video
9 for i, (x_audio, x_rgb, Y_{data}) in number_of_batch_examples:
10 X_concat = contatenate((x_audio, x_rgb), dim=2)
11 E_t = X_concat
12 Y_{t+1} = LSTM(E_t) ← Full algorithm located at Fig. 21
13 Forget Gate = $LTM_{t-1} * \sigma(W_i[STM_{t-1}, E_t] + b_f)$
14 Learn Gate = $n_t * i_t$
15 LTM_t = Forget Gate + Learn Gate
16 $STM_t = a_t * m_t$, More info at Fig. 21
17 $Y_{t+1} = LTM_t[:, -1, :]$ (Many-to-one [40])
18 Y_{t+1} ← Dropout (0.5)
19 Y_{t+1} = LSTM(E_t) ← Another LSTM (Bidirectional)
20 Y_{t+1} ← Dropout (0.5)
21 $Y_{t+1} \leftarrow \sigma(Y_{t+1})$
22 Loss = CrossEntropyLoss(Y_{T+1}, Y_{data})
23 Append Loss into an array of losses (graph out report)
24 Adam Optimizer (Y_{T+1})

Figure 11: Above is a latex representation of the first algorithm coded. The fully connected net is located within line 12 to 18 being trained inside a for loop of batch data.

RNN vs LSTM

Recurrent Neural Network (RNN), similar to a regular neural network, is a model that can compute sequential temporal patterns. They're computationally expensive compared to spatial algorithm since their memory intensive on the hardware usage [22, 34]. We need a model that can save more parameters, predict further outcomes in the future. Unlike RNN, LSTM has a cell state to control what parameters are relevant to preserve more extended sequential prediction [34]. If you remember Deep ResNets "Identity Mapping" [35], a cell state inside LSTM has the option to forget parameters. Thus preserve computation for other parameters that are relevant — saving more room for features that have more approximation between each label. With an identity map, we can "skip" parameters that are irrelevant to a sequential pattern. In other words, LSTM can predict the outcome further into the future compared to RNN through shortcuts. Which is a tradeoff because “shortcuts” require more usage from the hardware depending on how I develop my pipeline. We’ll see the performance gap in the experiment section. A sigmoid neural net layer controls these gates. The sigmoid function inside a forget gate outputs a number between 0-1: 0 for being irrelevant to the video label and one the opposite. The content with the highest score to 1 gets to save the next cell state. We utilize this model to see a series of video frames, and the first frame could be a series of cars on youtube, later classify a video game with race-cars like NASCAR: two different genres that need to be approximate. The dimensions for the input will be the same as RNN and Keras have an API function called keras.layers.

Forward LSTM combine with backward LSTM

LSTM alone focuses on future sequential predictions whereas bidirectional LSTM has an extra set of “directions” to compute future and past content. Current cell state in a single word LSTM doesn’t have enough content to compute the future. For example, if given the input sentence "Hi there, Teddy...", an LSTM needs to use three cell state for each word. The last cell receives the input word "Teddy" at time t=3, and the LSTM needs to decide if the sentence is either talking about a teddy bear or Teddy Roosevelt (president) before t=4. It’s not enough content to understand the sentence immediately. In the real world, we need more parameters at a time to predict future outcomes. Bidirectional LSTM can have one LSTM compute the first word at (t=1) and have another layer computing the last word. At (t=1,2,3) we have one directional LSTM computing the words "Hi there, Teddy" and backward LSTM computing backward "...becoming Mr. President", assuming if we have enough data, the Bi-LSTM has enough content to predict the next word after "Hi there, Teddy..." which is "...Roosevelt...". I plan on applying the same technique for a GRU.

CODED IN KERAS

To code this model in keras, we need to utilize the functional API again for "keras.layers.LSTM", except we provide the flag “go_backwards” to change the direction of the cell state. Each layer will go_backwards, return false or true back to replicate a bidirectional architecture. RGB and Audio will have their LSTM to

concatenate a fully connected dense, later sent to the final output. Back to the data, both RGB and audio from frame-level data will propagate into their LSTM models. RGB has the dimensions of (batchx100x1024): 100 serving the number of frames encoded from 100 seconds of video, and 1024 for the number of RGB features. As for audio frame-level dimensions, we have (batchx100x128): 100 completing some frames from 100 seconds of sound, and 128 for the number of audio features in one second. Once we have both features computed through their LSTM, the output is concatenated with a softmax approximation into a class label.

CODED IN PYTORCH

Each layer from the LSTM class in PyTorch has a bidirectional setting to set a flag true or false. If the flag is set to "true," LSTM model will add another LSTM layer computing following data backward and return hidden/output units from both models into a vector. The first half of the vector will have groups from hidden & output to make the first LSTM and the second half the second LSTM. The second LSTM with the bidirectional flag is set to "true," summing up to four LSTM. From my experience stacking up four layers will crash the server for an excessive amount of parameters to compute.

Stream LSTM (IDENTITY MAPPING + LSTM)

I’ve decided to create my own "Identity mapping" [35] implemented on top of LSTM. It'll be called LSTM stream. A combination of LSTM and a Resnet. There’s going to be two critical modules in this design: a fully connected dense layer and an LSTM. The first

output will lead to the next LSTM [22] and the second, into another fully connected layer to skip connections. There are hundreds of video frames I need to compute so the fully connected dense layers will help prevent a vanishing gradient: an overdose of knowledge for the computer. Remember we are dealing with a temporal model (huge memory demand). LSTM does the heavy lifting, fully connected dense layer distributes the work. In theory, both frameworks should have more work for the GPU and less on the CPU.

Algorithm 3: Stream LSTM is similar to Bidirectional except we have the pipeline seperate for the audio and rgb. No concatenation until we reached the sigmoid approximation which is the final output.

Input :
1 (b): Memory blocks bias
2 (W): Weights for the perspective layers
3 (STM): Short term memory initialize random parameters
4 (LTM): Long term memory initialize random parameters
5 Dimensions for frame-level x-inputs: (Batch x 100-Frames x Features)
6 x_audio: Compressed PCA frame-level content of audio (64 x 100 x 128)
7 x_rgb: Compressed PCA frame-level content of spatial footage (64 x 100 x 1024)
8 Y_{data}: genre labels for records in x data
Output: Y_{T+1} Classified genre label of a youtube video
9 for i, (x_audio, x_rgb, Y_{data}) in number_of_batch_examples:
10 X_concat = contatenate((x_audio, x_rgb), dim=2)
11 E_t = X_concat
12 Y_{t+1} = LSTM(E_t) ← Full algorithm located at Fig. 21
13 Forget Gate = $LTM_{t-1} * \sigma(W_i[STM_{t-1}, E_t] + b_f)$
14 Learn Gate = $n_t * i_t$
15 LTM_t = Forget Gate + Learn Gate
16 $STM_t = a_t * m_t$, More info at Fig. 21
17 $Y_{t+1} = LTM_t[:, -1, :]$ (Many-to-one [40])
18 Y_{t+1} ← Dropout (0.5)
19 Y_{t+1} = LSTM(E_t) ← Another LSTM (Bidirectional)
20 Y_{t+1} ← Dropout (0.5)
21 Loss = CrossEntropyLoss(Y_{T+1}, Y_{data})
22 Append Loss into an array of losses (graph out report)
23 Adam Optimizer (Y_{T+1})
24 Sigmoid Approximator (Y_{T+1})

Figure 12: Above is a latex representation of the first algorithm coded. The fully connected net is located within line 9 to 13 being trained inside a for loop of batch data.

CODED IN KERAS

Initialize the keras LSTM class to support incoming frame-level input for audio and RGB. The output decodes the data to have insight behind sound patterns in one-time direction. The next set of LSTM will decode the same data except the returning backward sequence (go_backwards=True), decoding the following data from the end of the temporal feature. The output vector is later distributed with a dense layer function to save computation runtime during optimization [35].

```
# LSTM
stream_lstm_1_x1 = LSTM(128, return_sequences=True, go_backwards=False, name='lstm_1_x1')(stream_fc_1_x1)
stream_lstm_1_x2 = LSTM(1024, return_sequences=True, go_backwards=False, name='lstm_1_x2')(stream_fc_1_x2)
# LSTM
stream_lstm_2_x1 = LSTM(128, return_sequences=True, go_backwards=True, name='lstm_2_x1')(stream_lstm_1_x1)
stream_lstm_2_x2 = LSTM(1024, return_sequences=True, go_backwards=True, name='lstm_2_x2')(stream_lstm_1_x2)
```

Figure 13 Above is only a snippet of my code. Full source is on my GitHub

CODED IN PyTorch

First I created a python class for the model, the input size needs to be referenced to create the hidden and cell state variables [22] since the algorithm in PyTorch needs to have more intuition of an LSTM. In Keras, the "hidden" variable and cell states provided inside the LSTM function. Each hidden & cell state is fed into the LSTM PyTorch model class setting bidirectional flag variable to “true.” PyTorch would return the vector for both forward and backward temporal outputs compared to keras. The rest is later sent into a dense layer to save computational space during optimization.

```
# Initialized in the model class
self.lstm_1 = nn.LSTM(512, hidden_dim_1, layer_dim, batch_first=True, bidirectional=True)
self.lstm_2 = nn.LSTM(512, hidden_dim_2, layer_dim, batch_first=True, bidirectional=True)
# Referenced the two lstm inside forward propagation function
lstm_audio, (hn_audio, cn_audio) = self.lstm_1(frame_audio_fc_1, (h0_audio, c0_audio))
lstm_rgb, (hn_rgb, cn_rgb) = self.lstm_2(frame_rgb_fc1, (h0_rgb, c0_rgb))
```

Figure 14 Above is only a snippet of my code. Full source is on my GitHub

Neural Net concatenated with a Stream LSTM

This model is a combination of a multi-class binary classifier and stream-LSTM to cover input data for both video & frame data. Both outputs are concatenated, meaning the probability between each label is the average between both models. Both Keras and PyTorch both had their own concatenate/merge class with little difference.

Algorithm 4: Neural Net (video level) concatenated with a Stream LSTM (frame level). It's a combination of Algorithm (1) and (3).

Input :
(b): Memory blocks bias
(W): Weights for the perspective layers
(STM): Short term memory initialize random parameters
(LTM): Long term memory initialize random parameters
x_audio_video: Compressed PCA video-level content of audio (64 x 128)
x_rgb_video: Compressed PCA video-level content of spatial footage (64 x 1024)
Dimensions for frame-level x-inputs: (Batch x 100-Frames x Features)
x_audio_frame: Compressed PCA frame-level content of audio (64 x 100 x 128)
x_rgb_frame: Compressed PCA frame-level content of spatial footage (64 x 100 x 1024)
Y_{data}: genre labels for records in x data
Output: Y_{T+1} Classified genre label of a youtube video
for i, (x_audio_video, x_rgb_video, x_audio_frame, x_rgb_frame, Y_{data}) in number_of_batch_examples:
    A_{T+1} = Neural Net (x_audio_video, x_rgb_video)
        $A_{T+1} \leftarrow$ Decoder Net ((x_audio_video, x_rgb_video), W_i)
        $A_{T+1} \leftarrow$ Feedforward (σ(x_audio_video, x_rgb_video), W_i))
        $A_{T+1} \leftarrow$ Rectifier Activated Function()
    E_{t_2} = x_rgb_frame
    E_{t_1} = x_audio_frame
    B_{t+1} = LSTM(E_{t_1}, E_{t_2}) $\leftarrow$ Full algorithm located at Fig. 21
        Forget Gate = $LTM_{t-1} * \sigma(W_i[STM_{t-1}, (E_{t1}, E_{t2})] + b_f)$
        Learn Gate = $n_t * i_t$
        LTM_t = Forget Gate + Learn Gate
        $STM_t = a_t * m_t$, More info at Fig. 21
        $B_{t+1} = LTM_t[:, -1, :]$ (Many-to-One LSTM technique [40])
    $B_{t+1} \leftarrow$ Dropout (0.5)
    B_{t+1} = LSTM(E_{t1}, E_{t2}) $\leftarrow$ Another LSTM (Bidirectional)
    $B_{t+1} \leftarrow$ Dropout (0.5)
    $Y_{t+1} \leftarrow \sigma$ (concat(A_{t+1}, B_{t+1}))
    Loss = CrossEntropyLoss(Y_{T+1}, Y_{data})
    Append Loss into an array of losses (graph out report)
    Adam Optimizer (Y_{T+1})

Figure 15: Above is a latex representation of the first algorithm coded. The fully connected net is located within line 9 to 13 being trained inside a for loop of batch data.

RESULTS (Experiment)

There were a total of eight deep learning algorithms experimented for this report: Deep Neural Net, Bidirectional LSTM, Stream LSTM, and a Neural Net concatenated with a Stream LSTM. I only mention four since I coded them in two different deep learning frameworks: keras and Pytorch, summing up to eight. My intention coding in two different AI frameworks is to analyze the relationship between the two, later decide which one I would use for research or production in the industry. I experimented more than once for each algorithm, then figure out what settings for the batch size, epochs, and learning rates are favorable.

Batch size: A subset of a data set to feed into the algorithm. And in this case a subset of training data. One batch would optimize the weight decisions (parameters) from the machine learning model. Later repeat the same cycle with the next batch of training data.

Epoch: One epoch means every batch from the training set has been sent to the machine learning model to train. If we had two epochs and three batches, the total number of iterations would be six. There go our parameters were updated six times.

Learning rate: The rate of how quick a model decides to abandon its old parameters. If the rate is low, the parameters will be able to lower the lost but at a slow pace. If the rate is high, the parameters will update but

steadily decrease & increase the performance arbitrarily for every epoch iteration.

Setup

Throughout my experiments, I've had multiple occasions where my server crashed from using an excessive amount of CPU & GPU separately during training. It's important to manage your hardware correctly for training. The question now is how I can monitor my hardware usage? After discovering wandb in November, a startup that's going public last year [27], they've open sourced an analytic machine learning tool to monitor my results: Record live training/test results of your lost, accuracy, and hardware usage during training.

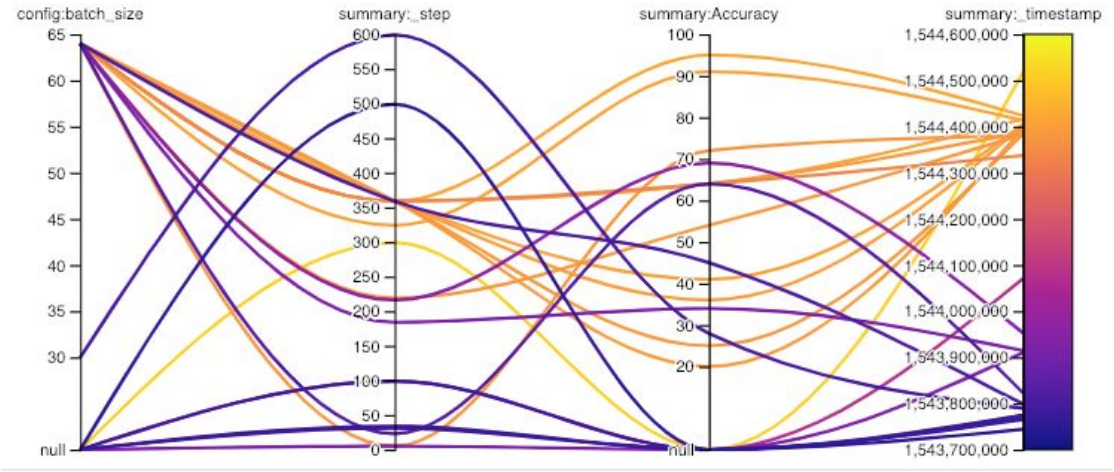


Figure 16 : Above is a parallel coordinate generated from wandb after training each experiment in PyTorch. (View my wandb account to interact with the graph)

Wandb was able to display parallel coordinates for most of my configurations (batch_size, epoch, learning rate). The graph above gives better intuition what values I should tune, enabling me to streamline multiple experiments. On my first days with

wandb, I forgot to set a tracker on my learning rate for every test in PyTorch. The ones I set a tracker are available on my wandb account (public). I have another parallel coordinate graph for my keras experiments which is visible under Appendix F. I plan on discussing three critical figures from my trials: accuracy performance, number of iterations to train, and memory management.

Training (PyTorch)

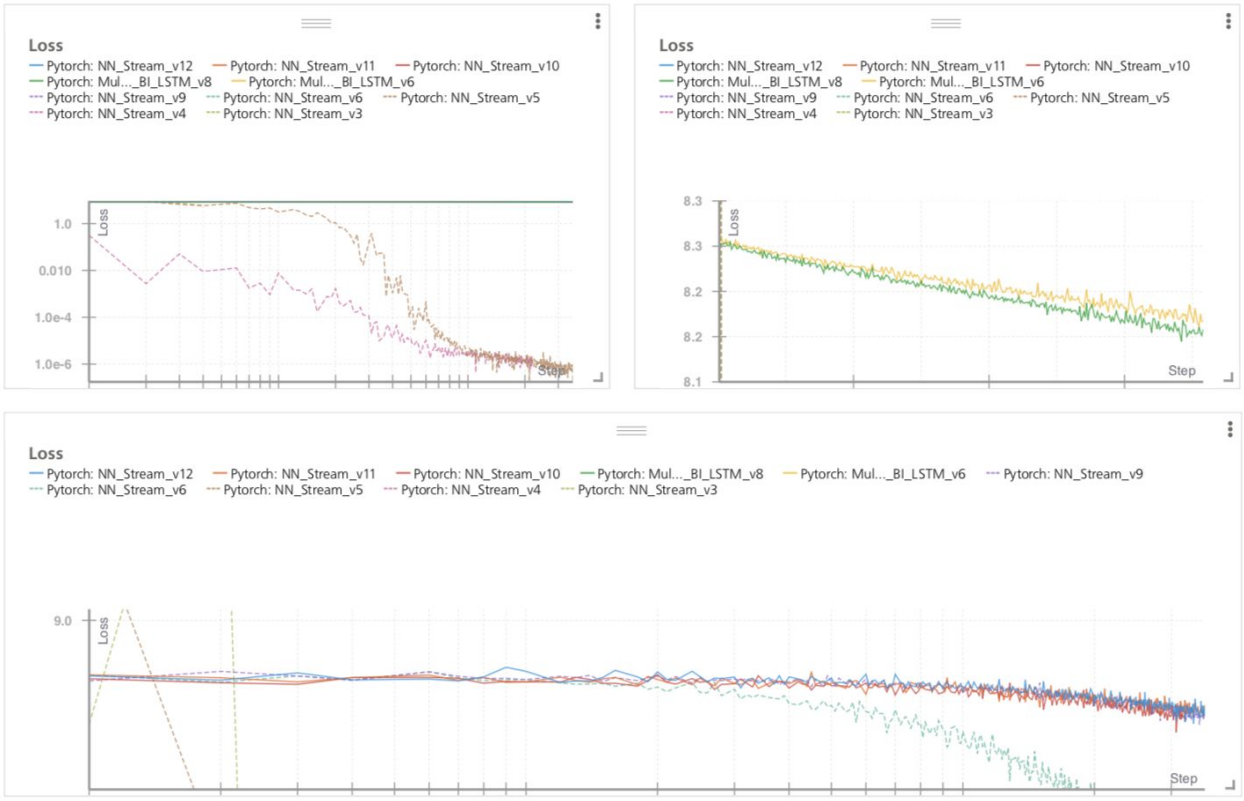


Figure 17: Training session for every PyTorch model I experimented on wandb
(View my wandb account to interact with the graph)

Initially, when you first start training your network, your lost function might have an immediate dip (above graph) due to randomly-initialized parameters. The gap between

the predicted output and the expected results is huge since casual settings are not well mapped. After a series of iteration from training, the loss output will start to have a consistent decay. Most of the algorithms I experimented (10 total), initially their loss was at 10%, now they dropped down to 1-9% after 300-500 epoch iteration. It looks like our model is improving. The lower the loss, the higher the chance each accuracy will have optimal performance.

Accuracy (PyTorch)

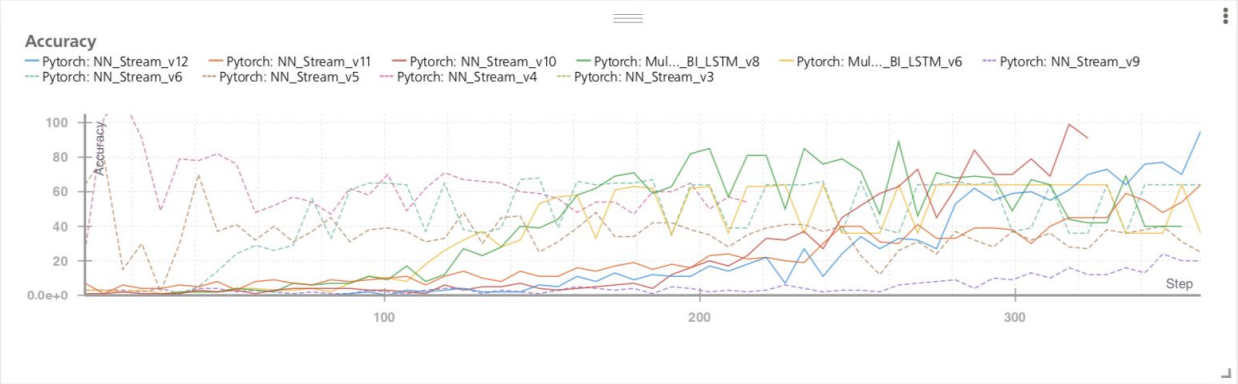


Figure 18: Accuracy session for every PyTorch model I experimented on wandb
(View my wandb account to interact with the graph)

Each measurement done for the accuracy had to compare two vectors (N Rows x 1 Column): a prediction vector against the expected target vector from training data. If both columns (output) from the vector are equal to each other, the accuracy would increase by 1%. Initially, the first iterations to calculate the accuracy will be unorthodox since the parameters are initially random. The settings will only update during training, but the “accuracy” for testing is the best metric performance in real practice. If the recommendation system from netflix is working flawlessly, the data scientist who can create the algorithm probably had excellent scores with their accuracy. Coming back to

our models, my performance for each algorithm starts to become consistent after 150 epochs, the same moment the lost function had a steady decay. The most significant factor in all of my designs was the number of hidden units in each model (method section) and the learning rate. Usually, values between 0.00045 and 0.03 were favorable for most of my experiments and any rate higher, or below the range, I provided will either take to long to train or have arbitrary performance. From all of my pytorch experiments, It looks like a neural net concatenated with a stream lstm (pytorch), had the best performance in terms of accuracy (95%) with only 300 epoch steps.

CPU and GPU usage (Pytorch)

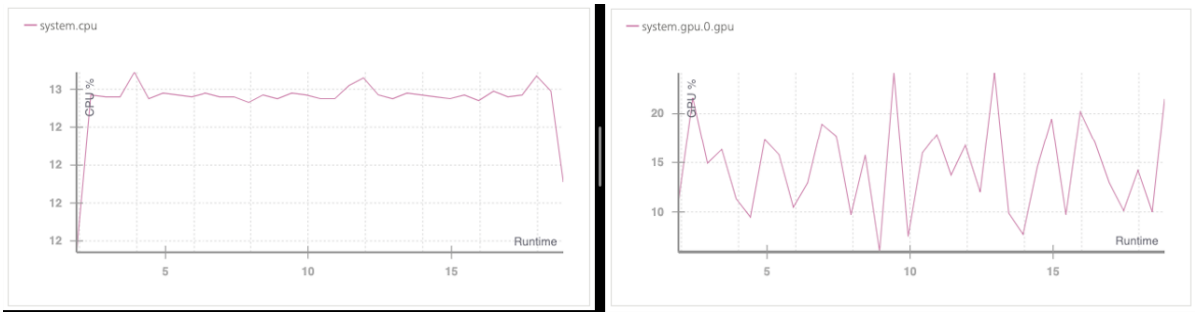


Figure 19: CPU/GPU session for a pytorch model I experimented on wandb
(View my wandb account to interact with the graph)

A neural net concatenated with an LSTM stream manages its hardware usage efficiently. It is running the algorithm quickly using a low usage on the CPU and GPU. The program would crash if the hardware used for either the CPU or GPU exceeded 100% yet both were under 25%. Based on the concatenated design of the model, the LSTM stream is using the CPU usage for recurrent memory usage while the neural net is dependent on the GPU. Since the model is using the CPU & GPU conservatively, it's

easily trainable and can deploy into production for an API cloud at a little cost. The amount of training was relatively low, making it cheap. The higher the CPU & GPU usage, the more expensive it cost.

Training (Keras)

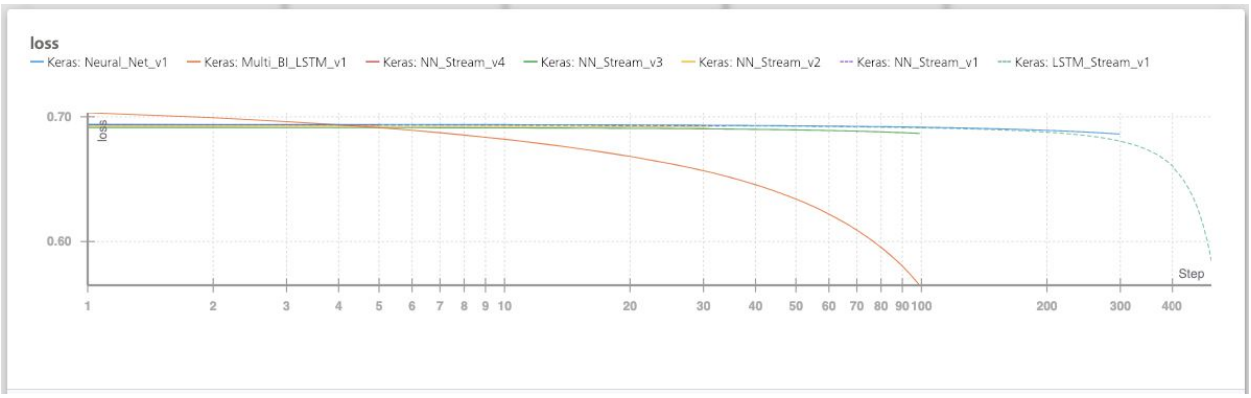


Figure 20: Training session for every Keras model I experimented on wandb
(View my wandb account to interact with the graph)

Most of my models initially trained at 70%, making the difference between PyTorch 1-9% results wider. Each model in keras took twice the amount of time compared to PyTorch to train. It's possible my current findings on hardware usages. Either keras is only using a CPU or GPU for each of my models trained, which explains the low run time. I notice another difference is the behavior between loss metrics. PyTorch had a noisy reaction while keras did not, I think keras has an API to filter out noisy behaviors during training and reserve values incrementing. Most models from keras started to crash after iterating more than 100, after taking the whole day, potentially crashing my server. It only takes an hour with PyTorch CPU & GPU usage to train 100 times. The outcome for training in keras will most likely transcend for accuracy as well.

Accuracy (Keras)

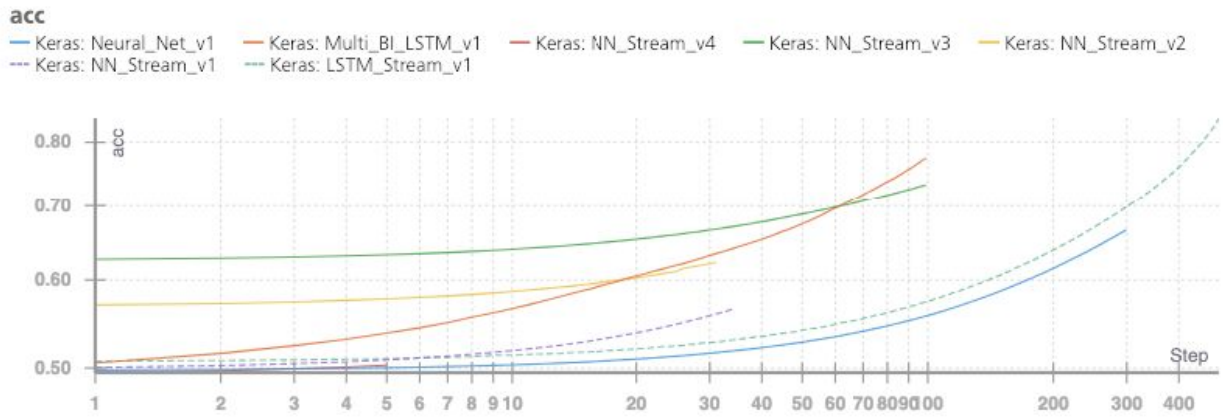


Figure 21: Accuracy session for every keras model I experimented on wandb
(View my wandb account to interact with the graph)

The accuracy for most models in keras started at 0.50 (50%), or higher, and later improve at a tedious pace. It's impractical for any algorithm to be 50% efficient with data it hasn't seen with random parameters initialized to extract information. Overall after a few iterations, every model improves their accuracy performance with most of them iterate at least 100 epochs. Any more epochs iterations in keras can crash, using an excessive amount of CPU & GPU. The metrics are not compulsive as PyTorch at the beginning of each iteration. After a series of experiments, the model with the highest accuracy in keras was a neural net concatenated with an LSTM stream. The number of epochs could potentially have higher efficiency, but the hardware would be an issue (mentioned later).

CPU and GPU usage (Keras)

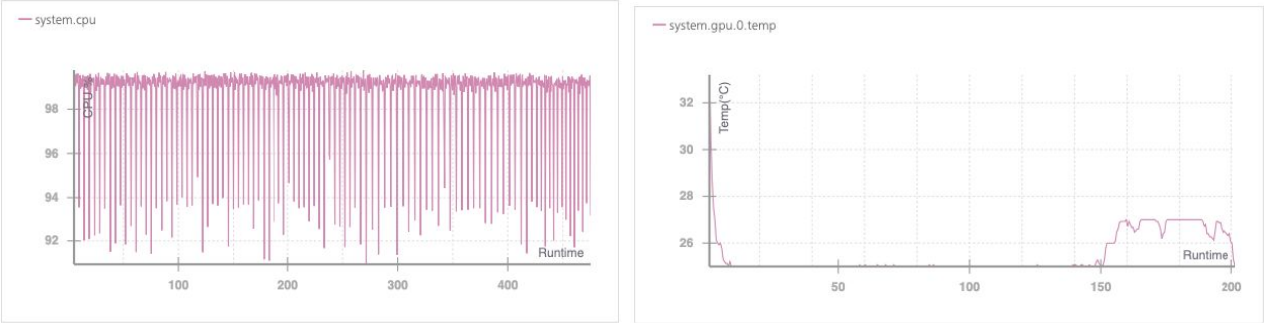


Figure 22: CPU & GPU session for a Keras model I experimented on wandb
(View my wandb account to interact with the graph)

All of the keras models didn't use the GPU but one of the experiments, also with the longest iteration without crashing, was used by a stream LSTM with 94% CPU usage and the temperature for the GPU rose up to 27% during training with no memory usage. The algorithm slowly begins to crash and throttle the GPU by accident at the server room. Throttling the GPU without using the memory for training is a red flag. A few more iterations and the server would've crash even though the LSTM barley exceeded pass 80% for accuracy. This model is unstable to train or also deploy onto an API cloud server for an excessive amount of CPU usage. I tried every protocol to allow GPU usage onto the framework [37] with no luck. This model would be expensive to deploy.

PyTorch vs Keras Overall test

Pytorch Report

	Loss	Accuracy	Learn Rate	Epoch	Batch Size	GPU Usage	CPU Usage	System Memory
Neural Net	1.75%	45%	0.01	300	30	29%	15.31%	13.89%
Multi-Bidirectional LSTM	7.89%	64%	0.003	300	64	55%	12.65%	12.26%
Stream LSTM	7.98%	64%	0.03	300	64	0%	97%	70.60%
Neual Net + Stream LSTM Concat	8.91%	95%	0.00045	300	64	24.13%	12.67%	16.49%

Keras Report

	Loss	Accuracy	Epoch	Batch Size	GPU Usage	CPU Usage	System Memory
Neural Net	68.5%	66%	300	84	0.13%	64.97%	14.37%
Multi-Bidirectional LSTM	56%	77%	300	64	0%	99.17%	31.19%
Stream LSTM	58.6%	84%	500	64	0%	93.37%	26.07%
Neual Net + Stream LSTM Concat	68.6%	73%	100	20	0%	96.84%	28.45%

Figure 23: CPU & GPU session for a Keras model I experimented on wandb. All of the experiments are available in Appendix b section.
(View my wandb account to interact with the graph)

After experimenting with both frameworks, I think keras is more straightforward to code but with less control to train between a GPU or CPU. As a stats major whose only concern with accuracy for research it's not an issue, which is why every experiment on keras for GPU usage almost got 0%. I tried most articles on how to use the GPU in keras [37], but none helped. If anything keras makes the GPU throttle if every algorithm runs on the CPU then magically use the GPU temperature. I can blame myself for not noticing a syntax I missed after coding or how I created my algorithm could be an issue. I recommended keras if your coding in deep learning for the first time to have more

intuition, later recreating the same code in PyTorch. Money is always an issue for any experiment and PyTorch helps me value my profit throughout the experience.

Concluding remarks and Future work

There are several ways to extend this project. If the dataset were uncompressed from Google researchers [1], we could use generative models (GAN) to create new youtube videos [28]. Video game graphic research or companies who wish to make a simulation could easily benefit. Another project is to set up another dataset, on top of [1], to classify copyright infringement or adult content based on video frames: this would require another set of a dataset to train, saving old parameters for this thesis can preserve info using transfer learning. Not a lot of deep learning researchers have been posting articles lately with imbalanced data on video frames. Youtube genres' like finance and real estate are not massively distributed compared to video games & cosmetics [1]. We can use synthetic minority over-sampling (SMOTE) [16] or Generative Adversarial Network (GAN) [29] to generate new data to enhance training sessions for the weak labels. Having less bias, including yt8m, is a promising area every data scientist needs for their research. Computer vision continues to become a valuable asset in action recognition genre labels is no different. In terms of production, my algorithm can hopefully help youtube with their recommendation system, find more related videos for the customer.

Bibliography

[1] Abu-El-Haija, Sami. “Youtube-8M: A Large-Scale Video Classification Benchmark.” Google. 2016.

[2] Tsang, SH. “Review: Alexnet, CaffeNet - Winner of ILSVRC 2012 (image classification)”.https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160

[3] O’Shea, Keiron. “An Introduction to Convolutional Neural Network.” Aberystwyth University. Dec, 2015.

[4] Krizhevsky, Alex. “ImageNet Classification with Deep Convolutional Neural Network.” Neural Information Processing System (NIPS). 2012.

[5] C. Feichtenhofer, A. Pinz, and A. Zisserman. “Convolutional two-stream network fusion for video action recognition.” CVPR, 2016.

[6] Jia, Chengcheng. “Stacked Denoising Tensor Auto-Encoder for Action Recognition with Spatiotemporal Corruptions.” IEEE. 2017.

[7] Kim, Minhoe. “Building Encoder and Decoder with Deep Neural Networks: On the way to Reality.” IEEE. 2018.

[8] Metz, Luke & Maheswaranathan, Niru. “Learning Unsupervised Learning Rule.” Google Brain. 2018.

[9] Denil, Misha. “Predicting Parameters in Deep Learning.” University of Oxford. 2013.

[10] Hetherly, Jeffrey. “Using Deep Learning to Reconstruct High-Resolution Audio.” https://blog.insightdatascience.com/using-deep-learning-to-reconstruct-high-resolution-audio-29deee8b7ccd

[11] Kelly, Brendan. “Deep Learning-Guided Image Reconstruction from Incomplete Data.” Arxiv. 2017.

[12] Kuleshov, Volodymyr. “Audio Super-Resolution Using Neural Nets.” International Conference on Learning Representation (ICLR). 2017.

[13] Goodfellow, Ian. “Understanding and Improving Interpolation in Autoencoders via an Adversarial Regularizer.” 2018.

[14] Kanska, Katarzyna and Golinski, Pawel. “Using Deep Learning for single Image Super Resolution.” https://deepsense.ai/using-deep-learning-for-single-image-super-resolution/

[15] Provost, Foster. “Machine Learning from Imbalanced Data Sets 101.” New York University. 2016.

[16] Chawla, Nitesh. “Smote: Synthetic Minority Over-Sampling Technique.” Technique. Journal of Artificial Intelligence Research. 2002.

[17] Oyelade, OJ. “Application of K-means Clustering algorithm for prediction of Students’ Academic Performance.” International Journal of Computer Science and Information Security(IJCSIS). 2010.

[18] I. Laptev, M. Marszalek, C. Schmid, and B. Rozenfeld. “Learning realistic human actions From movies.” CVPR, 2008.

[19] H.Wangand, C.Schmid. “Action Recognition with Improved Trajectories.” ICCV, 2013.

[20] Yeom, Samuel. “Privacy Risk in Machine Learning: Analyzing the Connection to Overfitting.” IEEE 31st Computer Security Foundations Symposium. 2018.

[21] Feichtenhofer, Christoph. “Convolutional Two-Stream Network Fusion for Video Action Recognition.” CVPR, 2016.

[22] Sherstinsky, Alex. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network.” Arxiv. 2018.

[23] Abbas, Alhabib. “Vectors of Locally Aggregated Centers for Compact Video Representation.” International Conference on Multimedia and Expo (ICME). 2015.

[24] Liu, Lingqiao. “Compositional Model Based Fisher Vector Coding for Image Classification.” IEEE Transaction on Pattern Analysis and Machine Intelligence. 2017.

[25] Richard, Alexander. “A Bag-of-Words Equivalent Recurrent Neural Network for Action Recognition. “ Arxiv from the University of Bonn. 2017.

[26] Olena. “GPU vs CPU Computing: What to choose?” Medium. 2018. https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4

[27] Ha, Anthony. “Weights & Biases raises $5M to build development tools for machine Learning”. Techcrunch Article. 2018. https://techcrunch.com/2018/05/31/weights-biases-raises-5m-to-build-development-tools-for-machine-learning/

[28] Vincent, James. “NVIDIA has created the first video game demo using AI-generated Graphics.” The verge. 2018. https://www.theverge.com/2018/12/3/18121198/ai-generated-video-game-graphics-nvidia-driving-demo-neurips

[29] Mueller, Franziska. “GANerated Hands for Real-Time 3D Hand Tracking from Monocular RGB.” CVPR. 2018

[30] Liu, Dan-Ching. “A Practical Guide to ReLU.” Medium article. 2017. https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7

[31] Li, Fei-Fei. “Neural Networks Part 1: Setting up the Architecture.” CS 231 Convolutional Neural Networks for Visual Recognition”.

[32] Hyndman, Rob. “How to choose the number of hidden layers and nodes in a feedforward neural Network.” Stack exchange website. https://stats.stackexchange.com/questions/181/how-to-choose-the-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw

[33] Vazquez-Reina, Amelio. “Why are non zero-centered activation functions a problem in Backpropagation?” Stack exchange website. https://stats.stackexchange.com/questions/237169/why-are-non-zero-centered-activation-functions-a-problem-in-backpropagation

[34] Chung, Junyoung. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” NIPS. 2014

[35] He, Kaiming. “Deep Residual Learning for Image Recognition.” ILSVRC. 2016.

[36] Zagoruyko, Sergey and Chintala, Soumith. “A MultiPath Network for Object Detection.” Facebook AI Research (FAIR). 2016. https://www.youtube.com/watch?time_continue=2&v=0eLXNFv6aT8

[37] AEndrs. “Low GPU usage by keras / tensorflow?” Stackoverflow discussion. 2017 https://stackoverflow.com/questions/44563418/low-gpu-usage-by-keras-tensorflow

[38] Krishnan, Gokula. “Difference between the Functional API and the Sequential

API". Google group discussion. 2016.
https://groups.google.com/forum/#!topic/keras-users/C2qX_Umu0hU

[39] Peterstone. "Saving an Object (Data persistence)." Stackoverflow discussion. 2010.
https://stackoverflow.com/questions/4529815/saving-an-object-data-persistence

[40] Lu, Milo. "How can we define one-to-one, one-to-many, many-to-one, and many-to-many lstm neural networks in keras? [duplicate]." Stackoverflow discussion. 2018

[41] Gal, Yarin. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." NIPS Conference. 2018

[42] Silver, David. "Mastering the game of Go with Deep Neural Nets with Tree Search". Nature Internal Journal of Science. 2016

[43] Silver, David. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". DeepMind. 2017

[44] Ray, Tiernan. "Fast.ai's software could radically democratize AI". zdnet. 2018
https://www.zdnet.com/article/fast-ais-new-software-could-radically-democratize-ai

[45] Johnson, Khari. "Facebook launches Pytorch 1.0 with Integrations for Google Cloud, AWS, and Azure Machine Learning". venturebeat.com.
https://venturebeat.com/2018/10/02/facebook-launches-pytorch-1-0-integrations-for-google-cloud-aws-and-azure-machine-learning/?fbclid=IwAR0ZbFcn9U-plAx5uiKEsbosACSTjvoNruQsJkesgRbbqSHYx67Mu2M7_YE

[46] Philipp Schmidt, kaggle,
https://www.kaggle.com/philschmidt/youtube8m-eda

[47] NLP's ImageNet moment has arrived, 2018
http://ruder.io/nlp-imagenet/

[48] T.Q. Chen, Ricky and Rubanova, Yulia. "Neural Ordinary Differential Equations", Neuralps Conference. 2019.

[49] Talby, David. "Why Machine Learning Models Crash and Burn In Production". Forbes.com.https://www.forbes.com/sites/forbestechcouncil/2019/04/03/why-machine-learning-models-crash-and-burn-in-production/#48dd10272f43

[50] Makadia, Mitul. "Top 8 Deep Learning Frameworks". Dzone.com
https://dzone.com/articles/8-best-deep-learning-frameworks

[51] Aggarwal, Keshav. “A brief guide to Tensorflow Eager Execution”. Medium.com https://towardsdatascience.com/eager-execution-tensorflow-8042128ca7be?gi=64b7427864b6

[52] ArthurDent (anonymous username). “Keras Multiple ‘Softmax’ in last layer possible”. Datascience.stackexchange.com article. https://datascience.stackexchange.com/users/40180/arthurdent

[53] Sunnak, Abhishek. “Evolution of Natural Language Generation”. Medium Article https://medium.com/sfu-big-data/evolution-of-natural-language-generation-c5d7295d6517

[54] Madsen, Andreas. “Visualizing memorization in RNNs.” Distill journal 2019. https://distill.pub/2019/memorization-in-rnns/

[55] Rosenblatt, Frank. “The Perceptron: A Probabilistic Model for information storage and organization in the brain”. Cornell Aeronautical Laboratory. 1958. https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf

[56] Geirhos, Robert. “Imagenet-Trained CNNs are biased towards texture; increasing shape bias improve accuracy and robustness”. University of Tubingen ICLR 2019.

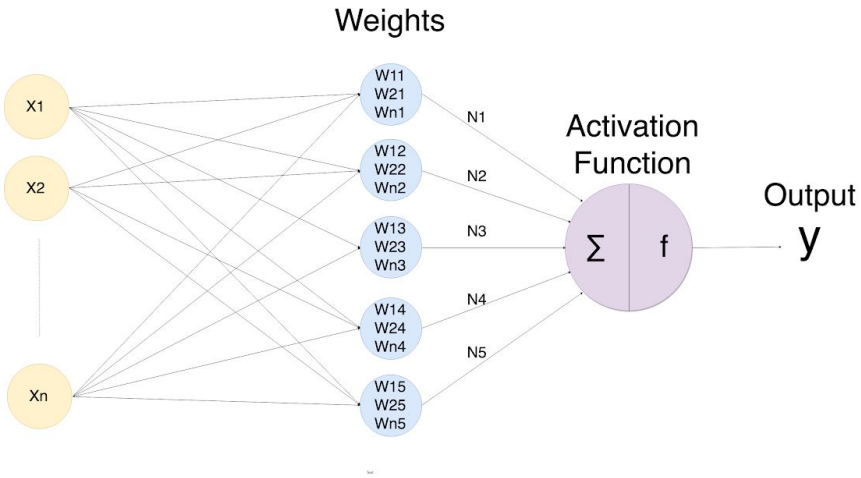
[57] Jeremy Howard: Deep Learning Frameworks - TensorFlow, PyTorch, fast.ai | AI Podcast Clips. https://www.youtube.com/watch?v=XHyASP49ses

Appendix A: Math Explained with Data Aggregated

This section explains the mathematics, and pseudo code (no framework), for each algorithm including what was used to measure: training, evaluation, hyper-parameter

tuning, and prediction. Each model drawn below will only be a small representation to understand the math. Full architecture is available on my appendix diagrams drawn in UML or on Tensorboard.

Neural Net



	N1	N2	N3	N4	N5
X1	W11	W12	W13	W14	W15
X2	W21	W22	W23	W24	W25
Xn	Wn1	Wn2	Wn3	Wn4	Wn5

Figure 24: Above we have a neural net structure of our yt8m data ‘x’ aggregate by weight parameters ‘w.’

An example of a neuron showing the input (x1-xn), their corresponding weights (w1-wn), and the activation function f applied to the weighted sum of the data. You also see a table representing the aggregation between the input and the weight decisions. It’s a higher level intuition how computation in linear algebra (matrix multiplication). Yt8m video level data will be the input while the weighted sums ‘w’ multiply with their respective amounts.

Scoring input of the model:
input * weight = guess

$$y = activation(max(0, f(\sum_{i=1}^{n} x_i w_i))) \tag{1}$$

$$y = activation(max(0, f(w_1 x1 + w_2 x2 + w_n x_n))) \tag{2}$$

$$max \tag{3}$$

$$f \quad \text{A logistical unit multipled with input and weight} \tag{4}$$

$$\sum_{i=1}^{n} \quad \text{Summation of all training examples from input} \tag{5}$$

$$x_i \quad \text{video-level input from yt8m} \tag{6}$$

$$w_i \quad \text{Weight size corresponding to the number of training example} \tag{7}$$

Figure 25: Above we have the formulas used for in a multi-class neural net. Above was coded on a separate latex file.

Above is the predicted output from the rectifier layer unit. It's one of the many activations to be part of a neural network to approximate signals from the input. If the data is negative, relu returns a 0; otherwise, the output of the signal returns the same value from the input. Any value greater than 0 activates a message to send more information to the next depth of neurological comprehension.

Deep Neural Net

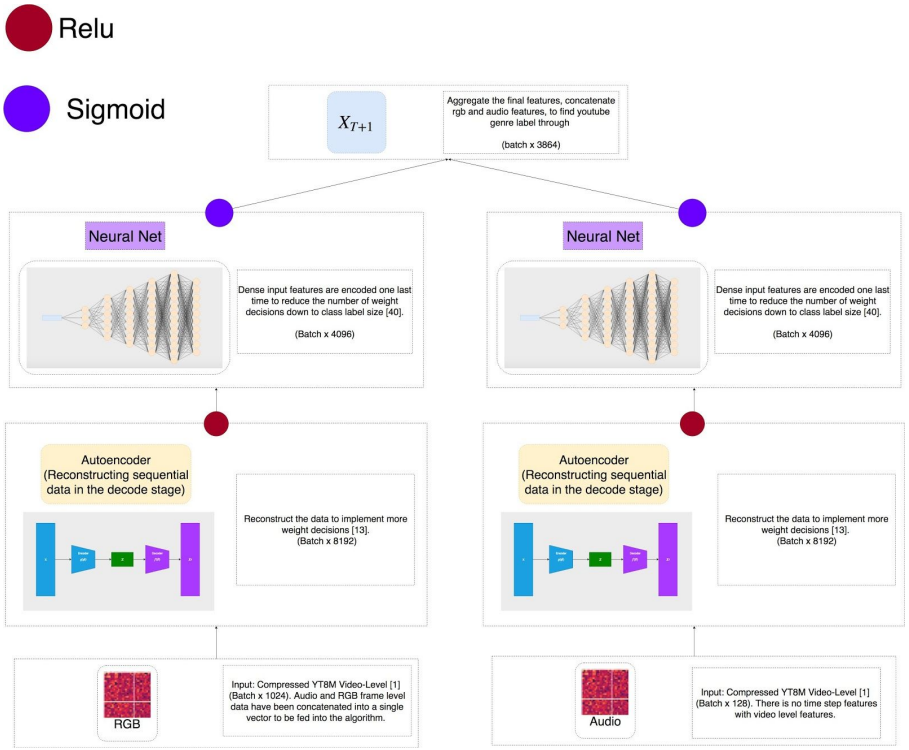


Figure 26: Above is a deep neural net used for the first series of experiments for easy design. Diagram was drawn using draw.io

Forward propagation

Predicted output $X_{T+1} = Concatenated(A + B)$

$$A = Neural\ Net(Autoencoder(RGB\ Input)$$

$$B = Neural\ Net(Autoencoder(Audio\ Input)$$

The first algorithm is really simple compared to the other three mentioned later in the book. We have the data for both rgb and audio video-level data fed onto the decoder, later increase the number of weight decisions to address compressed latent size smaller than the class label. Red signals are the relu activation function to approximate decisions between the input and aggregated weight decisions. Sigmoid is more

expensive to activate signals compared to relu during optimization, however for the last output, we use a sigmoid to

Gradient Descent

The difference between network predictions and the expected label from the data is the error. The network measures that error, and walks the error back over its model, adjusting weights to the extent they contributed to the error.

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

$$w = w - \sigma * \frac{dz}{dx}$$

$$\text{adjustment} = \text{error} * \text{weight's contribution to error}$$

In a feedforward network, the relationship between the neural net's error and a single weight will look like the following.

$$\frac{d_{error}}{d_{weight}} = \frac{d_{error}}{d_{activate}}\frac{d_{activate}}{d_{weight}}$$

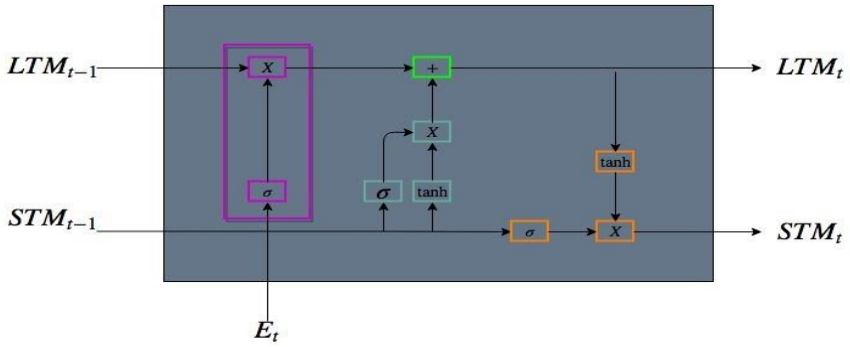
Figure 27: Above we have the optimization process used to update the weights. This was formatted on my private latex to screenshot.

Given two variables, error and weight, activation is a variable intervening the two. Activation is responsible for approximating the probability between labels in a given data set [33]. Change in weight affects activation and transformation inactivation affects error. Variables for activating and weight are differentiable to modify the error. Below is the ideal function to enable signals.

1. The limitation for our activated output needs to be in between (-inf, inf). For example, a sigmoid is only between zero and one whereas a linear function (ax + b) limit is (-inf, inf).
2. Non binary activations (step functions): intermediate activation values: And output continuous values in between (ex: sigmoid between -2 to 2). Sigmoid, tanh, and relu are all non-binary activations. Any decision that is not binary (discrete between two numbers)
3. Differentiable (vanishing gradient): Make sure the gradient is non-zero and recover during training.
4. Needs to be non-linear to replicate high dimensional mapping between data points. If every section of the model had linear activation, the model wouldn't be sophisticated enough to have a stack of signals.

Long-Short Term Memory

LSTM = LTM + STM
Long-Short Term Memory



Learn Gate

Previous time step for short term content is combined with current event (input) through two different activation function, later dot multiplied together.

Used to combine short term (previous time step) and current event through tahn activation:

$$n_t = \tanh(W_n[STM_{t-1}, E_t], +b_n)$$

Used to approximate what content to ignore after combining the two inputs:

$$i_t = \sigma\,(W_i[STM_{t-1}, E_t] + b_i)$$

Dot product to find Learn Gate:

$$\text{Learn Gate} = n_t * i_t$$

Forget Gate

Previous time step for short term content is combined with current event (input) through one activation function, afterwards perform dot product multiplication.

Combine current event and previous time step for short term content:

$$f_t = \sigma(W_f[STM_{t-1}, E_t] + b_f)$$

Dot product to find Forget Gate:

$$\text{Forget Gate} = LTM_{t-1} * f_t$$

$$\text{Forget Gate} = LTM_{t-1} * \sigma(W_f[STM_{t-1}, E_t] + b_f)$$

Remember Gate (Current time step for Long Term Memory)

Combine Learn gate (From short term memory with previous time step & current event) and Forget gate (Previous time step for long term memory). This will make the newest longest time step content.

$$LTM_t = \text{Remember Gate} = \text{Forget Gate} + \text{Learn Gate}$$

$$LTM_t = LTM_{t-1} * f_t + N_t * i_t$$

Use Gate (Current time step for Short Term Memory)

It uses long term memory that just came out of the forget gate and short term memory that came out of the Learn gate to come up with a new short term memory and an output.

The first input is the remember gate passed into the activation function tanh:

$$a_t = \tanh(W_a * \text{Remember Gate} + b_a)$$

Second input is used to approximate what content to ignore after combining short term from previous time step and current event:

$$m_t = \sigma\,(W_m[STM_{t-1}, E_t] + b_m)$$

Final output for new current short term step:

$$STM_t = a_t * m_t$$

Figure 28: Above is a formula sheet I made of a LSTM architecture. We have the following: Learn, forget, remember, and use gate.

Multi-Bidirectional LSTM

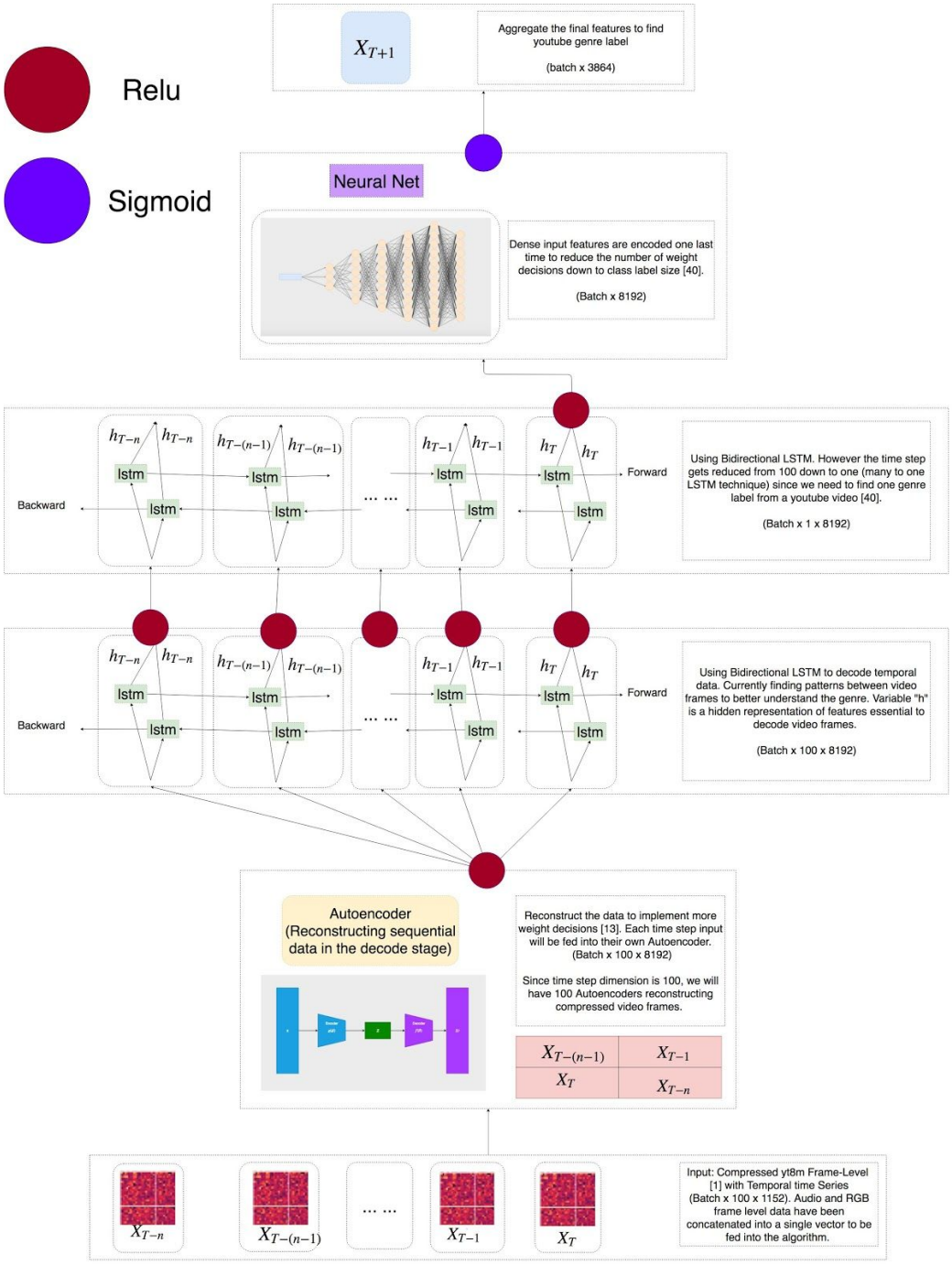


Figure 29: Above we have a bidirectional lstm created on draw.io

Stream LSTM

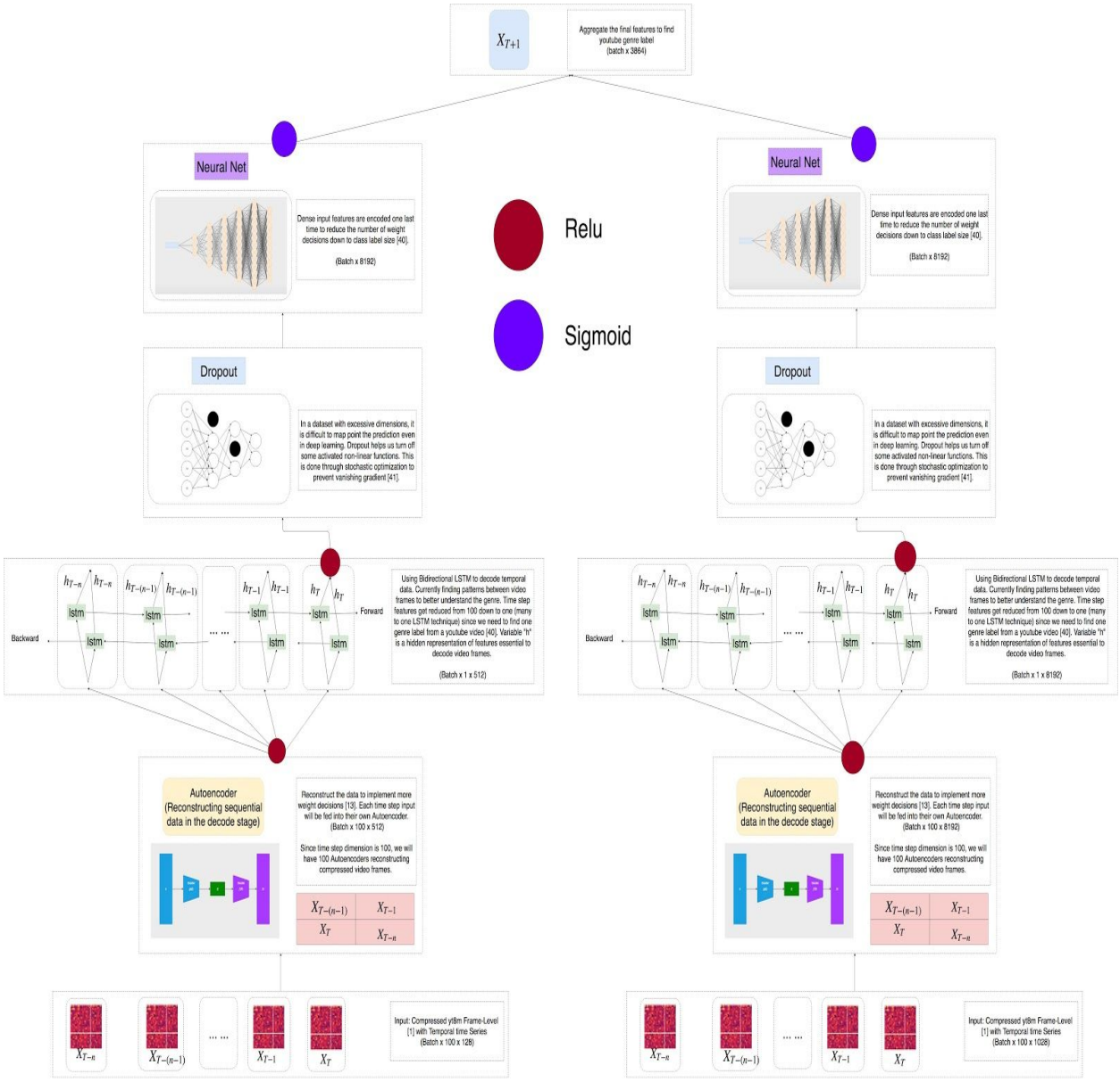


Figure 30: Above we have a stream lstm created on draw.io

Neural Net LSTM Stream

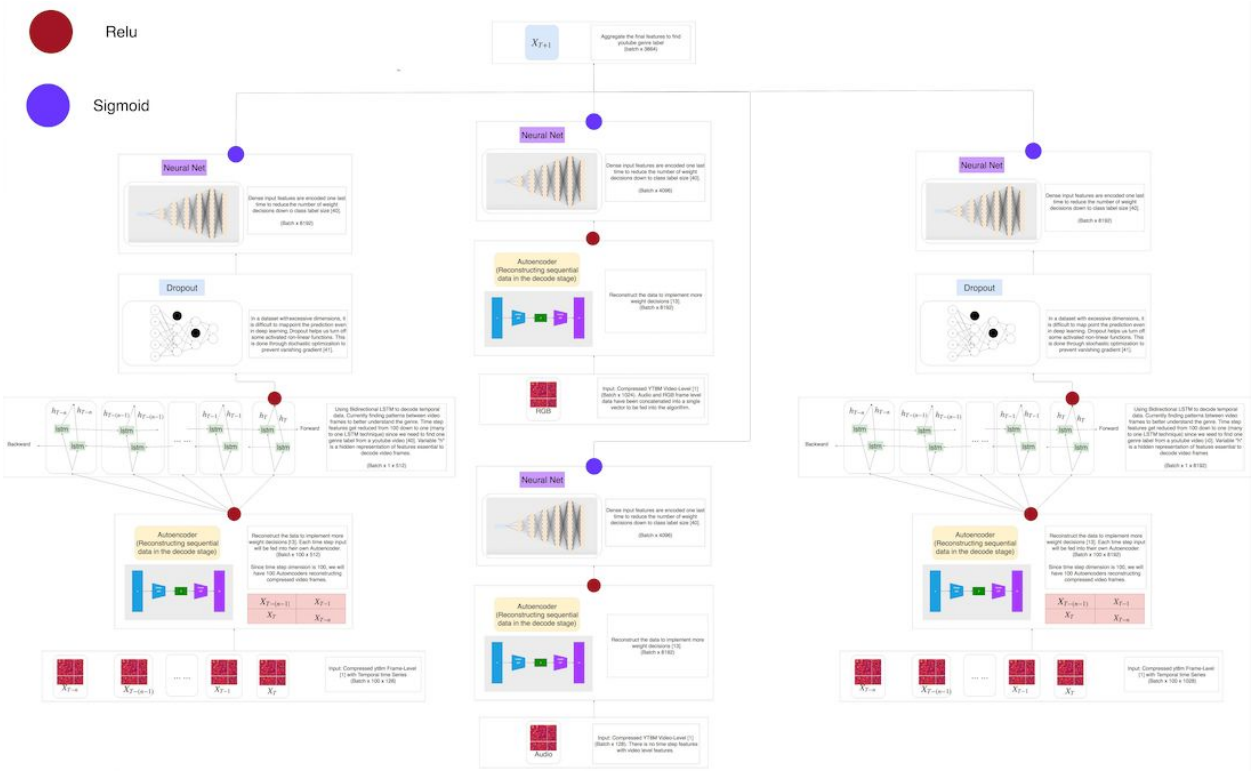


Figure 31: A Neural_Net_LSTM_Stream created on UML (draw.io). This is the final design for my algorithm and it's a lot to process all on one image. Some of the text is not visible so you will need to visit my github portfolio to see the entire diagram.

Appendix B: All of my wandb lost & accuracy experiments done with PyTorch

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

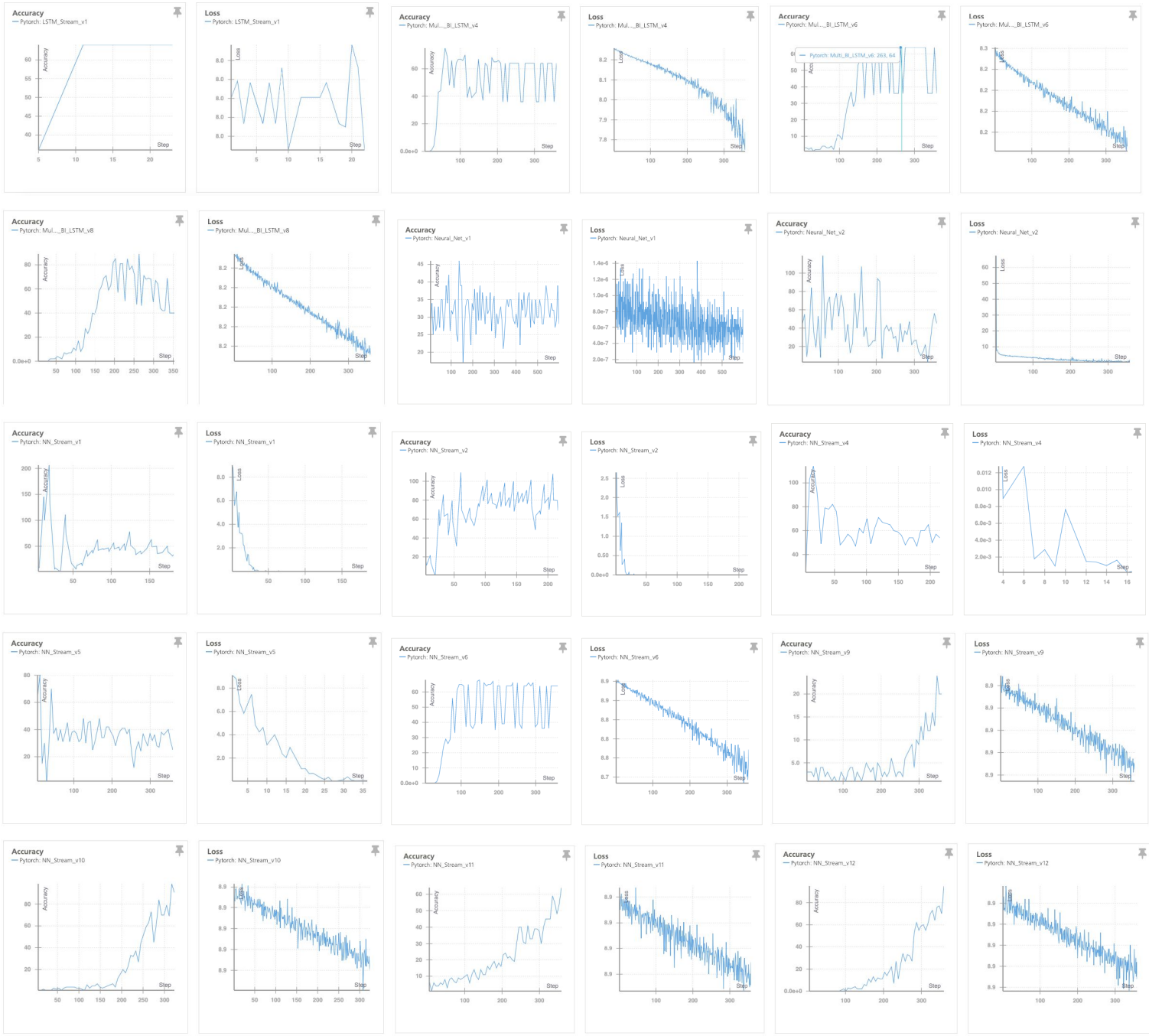


Figure 32: Above are lost and accuracy experiments done in wandb

Appendix C: All of my wandb CPU experiments done with PyTorch.

The three areas that are important for a machine learning model is accuracy, training time, and memory usage All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

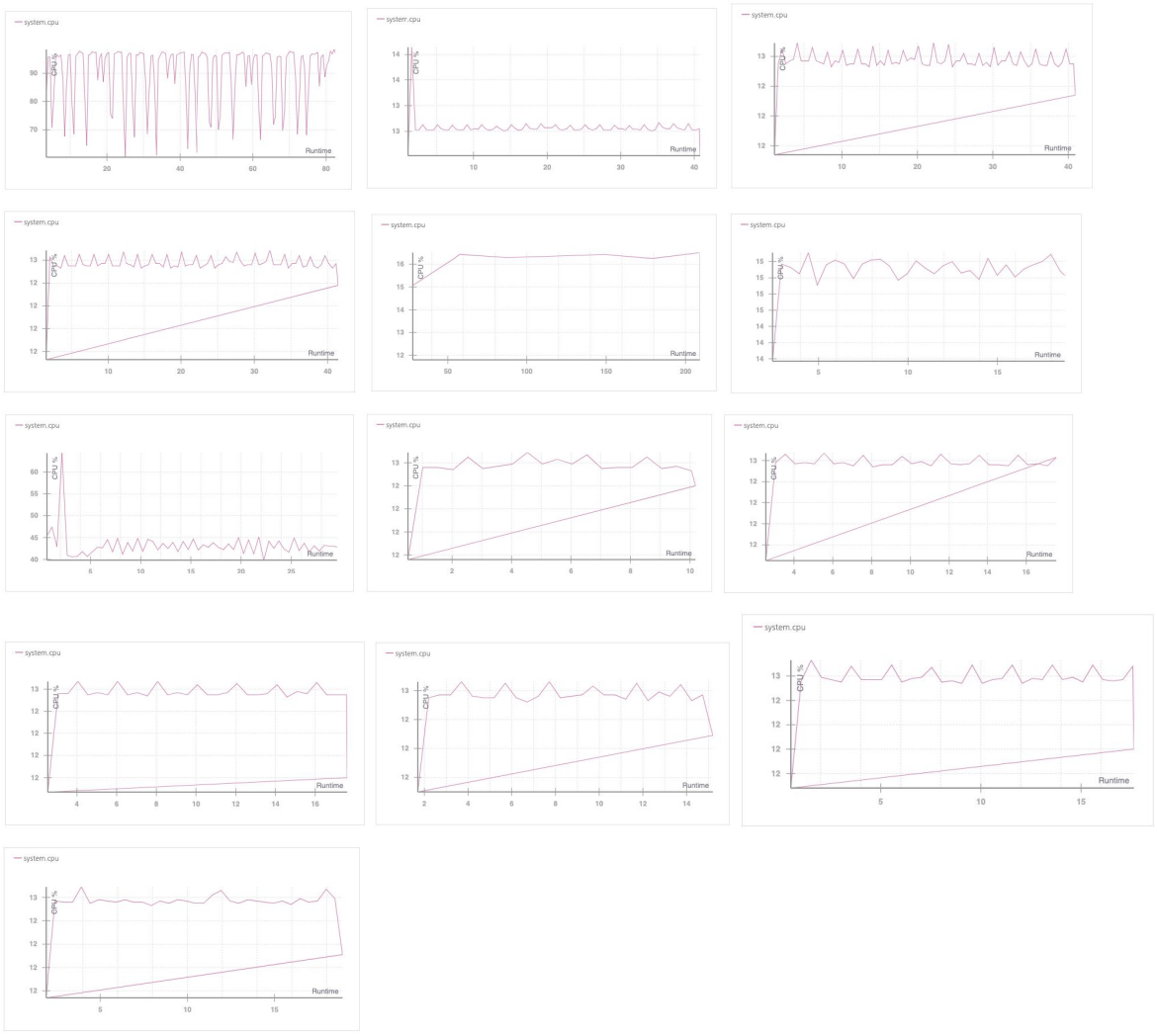


Figure 33: Above are CPU experiments done in wandb

Appendix D: All of my wandb GPU experiments done with PyTorch.

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

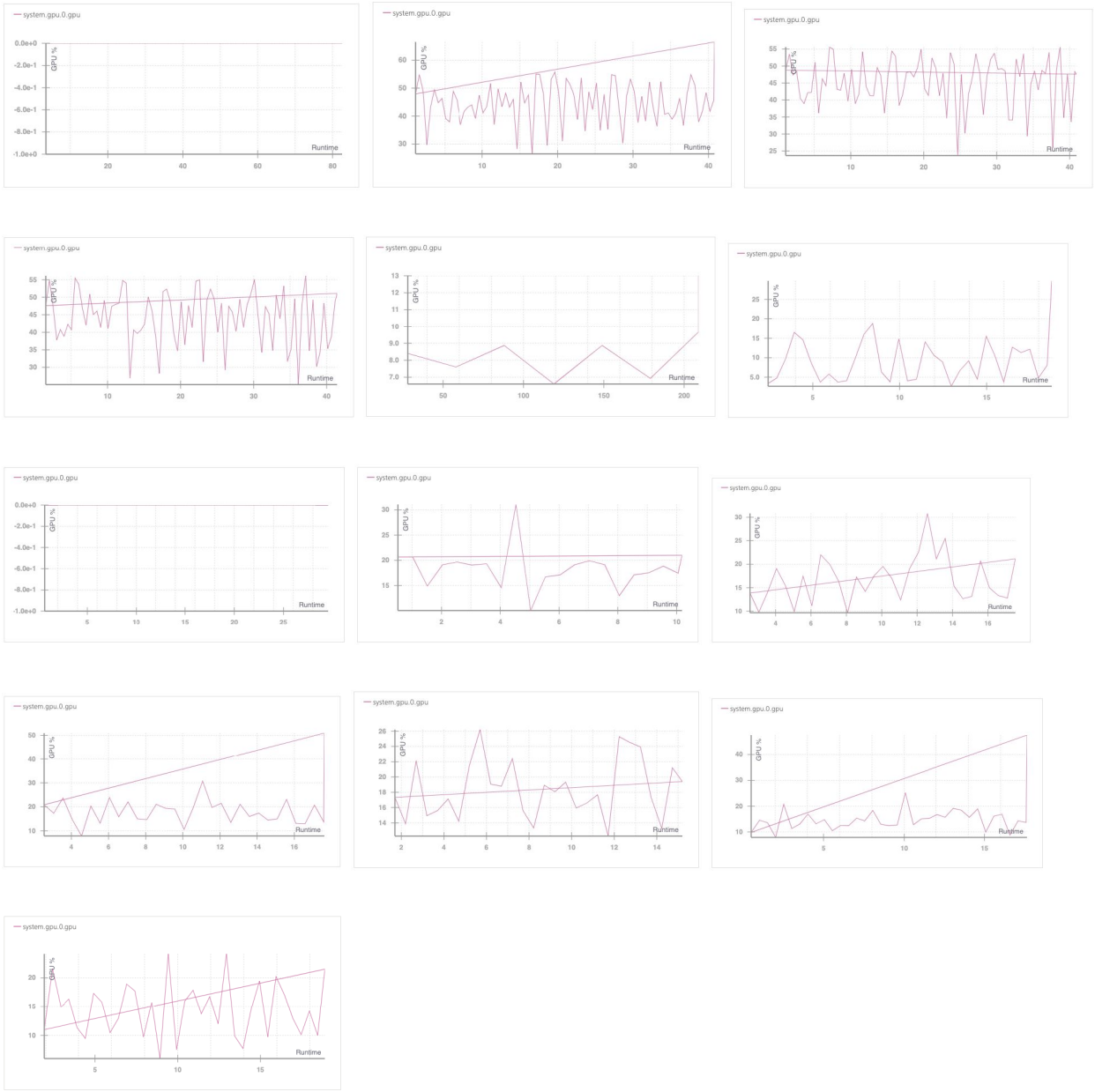


Figure 34: Above are GPU experiments done in wandb

Appendix E: All of my wandb hardware usage experiments done with PyTorch.

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

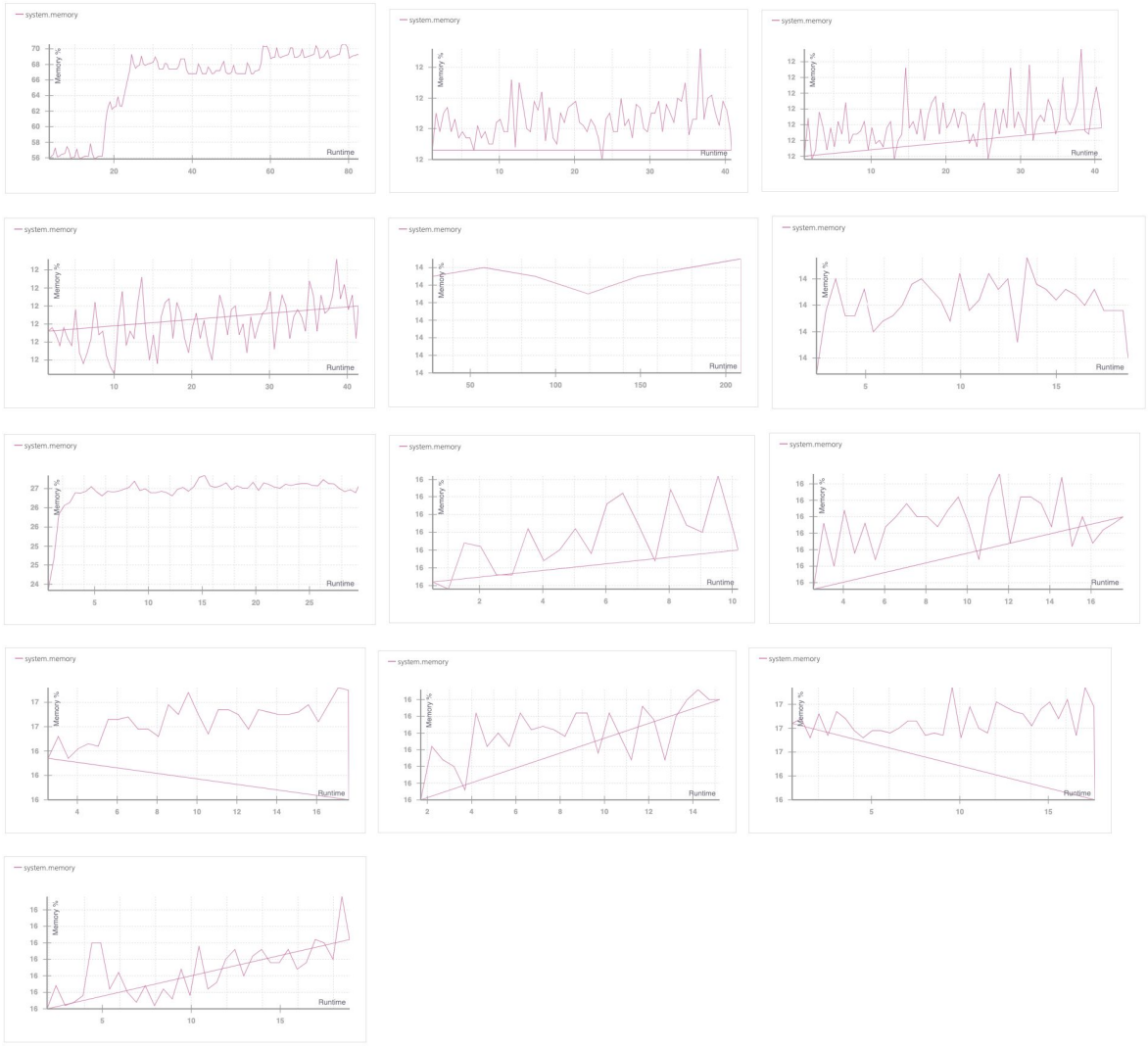


Figure 35: Above are hardware usage experiments done in wandb

Appendix F: All of my wandb lost & accuracy experiments done with Keras

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

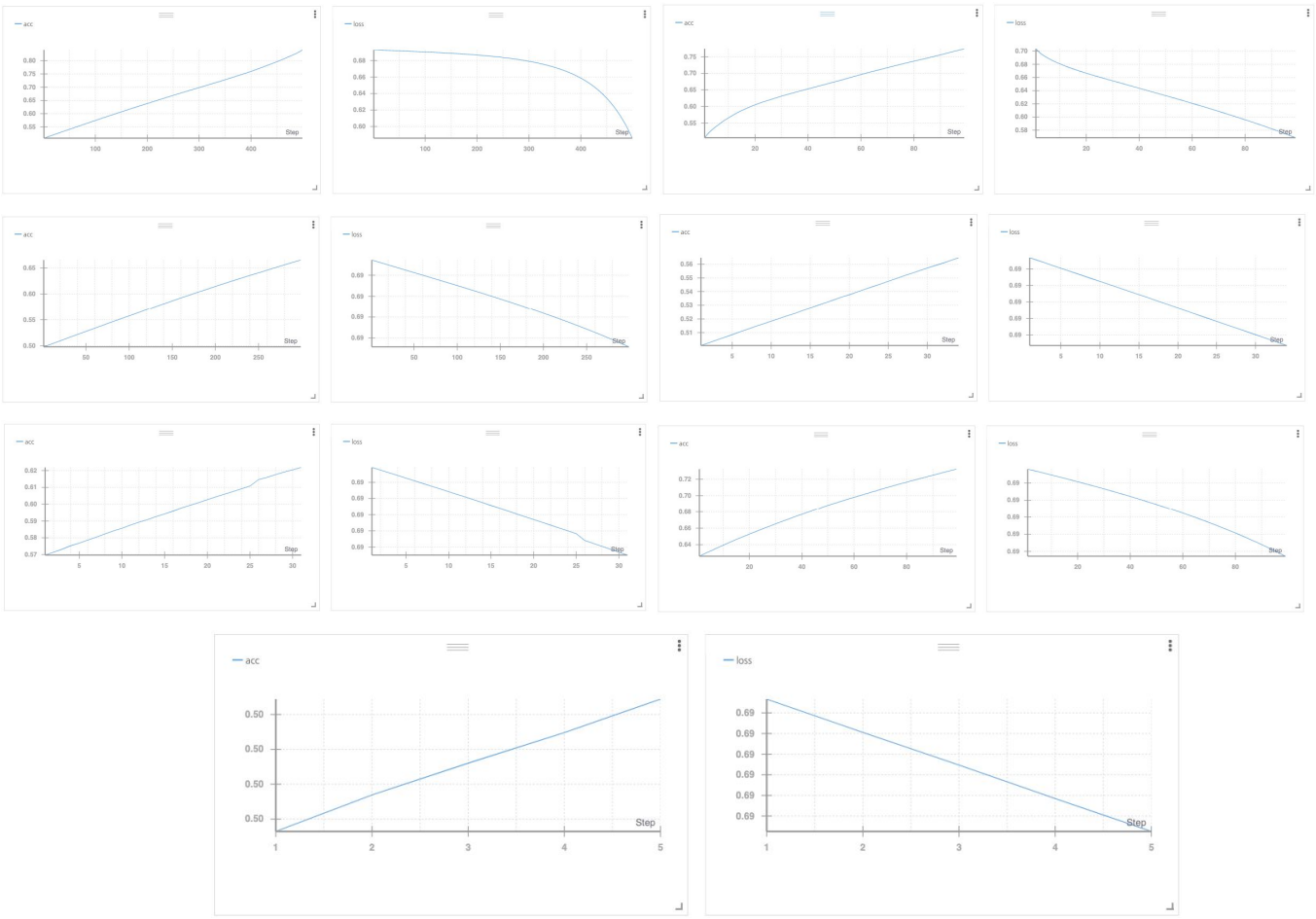


Figure 36: Above are my lost & accuracy (Keras) experiments done in wandb.

Appendix G: All of my wandb system experiments done with Keras

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

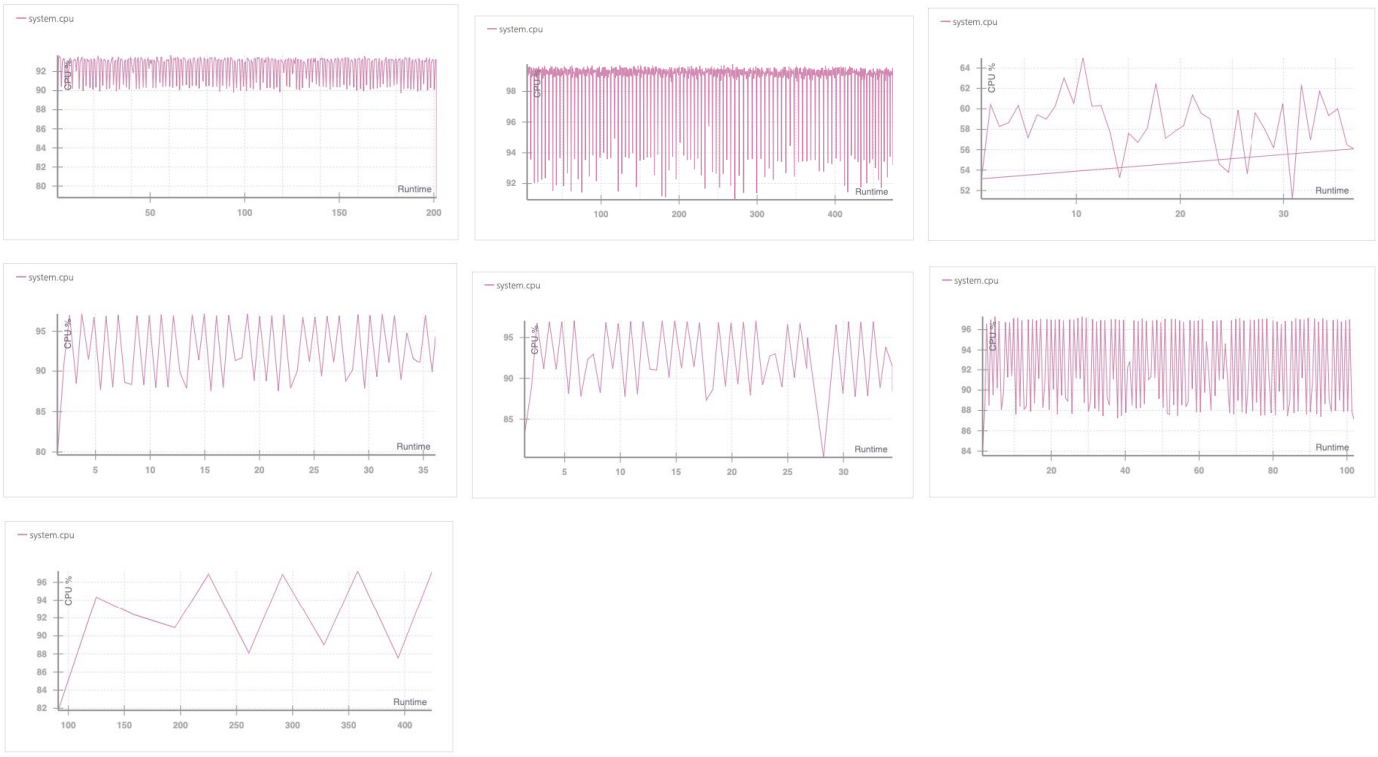


Figure 37: Above are system usage (Keras) experiments done in wandb.

Appendix H: All of my wandb GPU experiments done with Keras

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

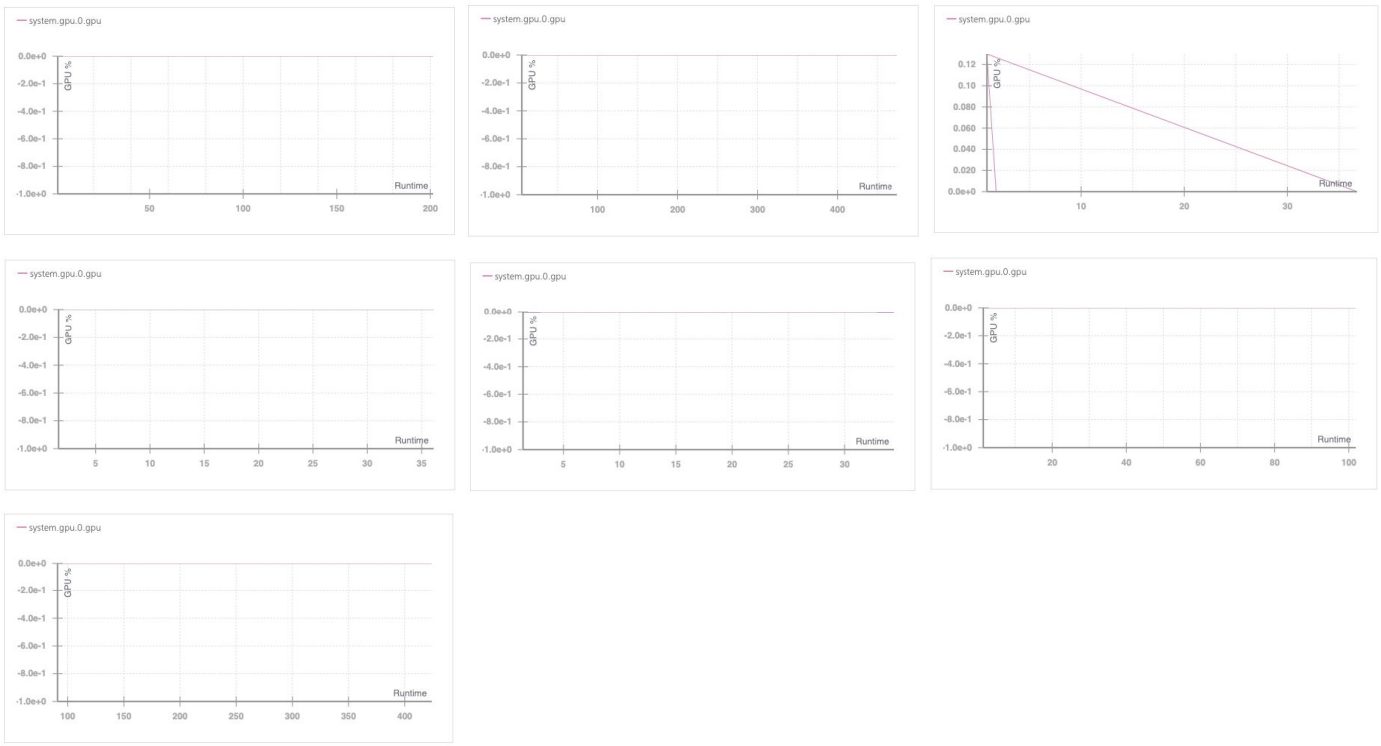


Figure 38: Above are GPU (Keras) experiments done in wandb.

Appendix I: System memory hardware experiments done with Keras

All of my experiments are available on my github or go on my wandb account. My most efficient algorithms are under results (experiment) section.

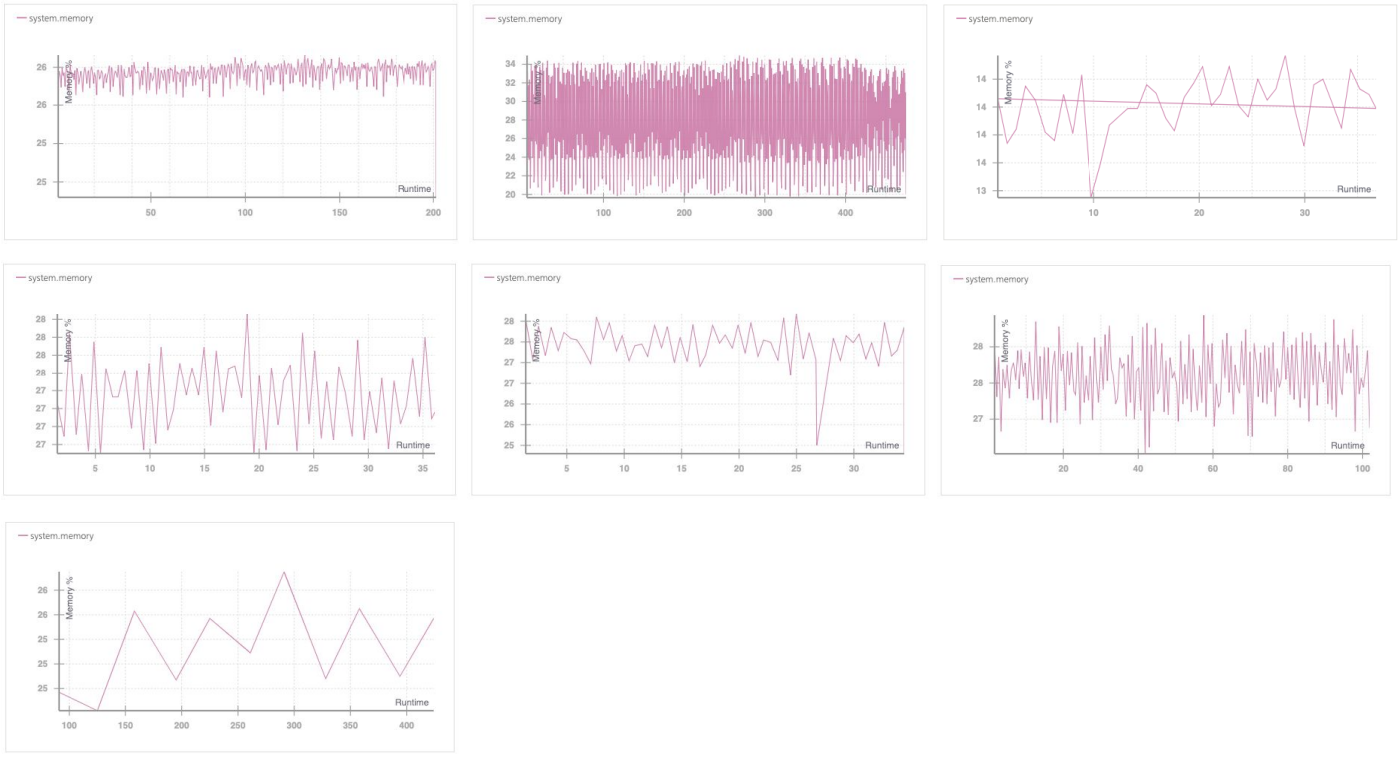


Figure 39: Above are system memory (Keras) experiments done in wandb.

Deep Neural Net

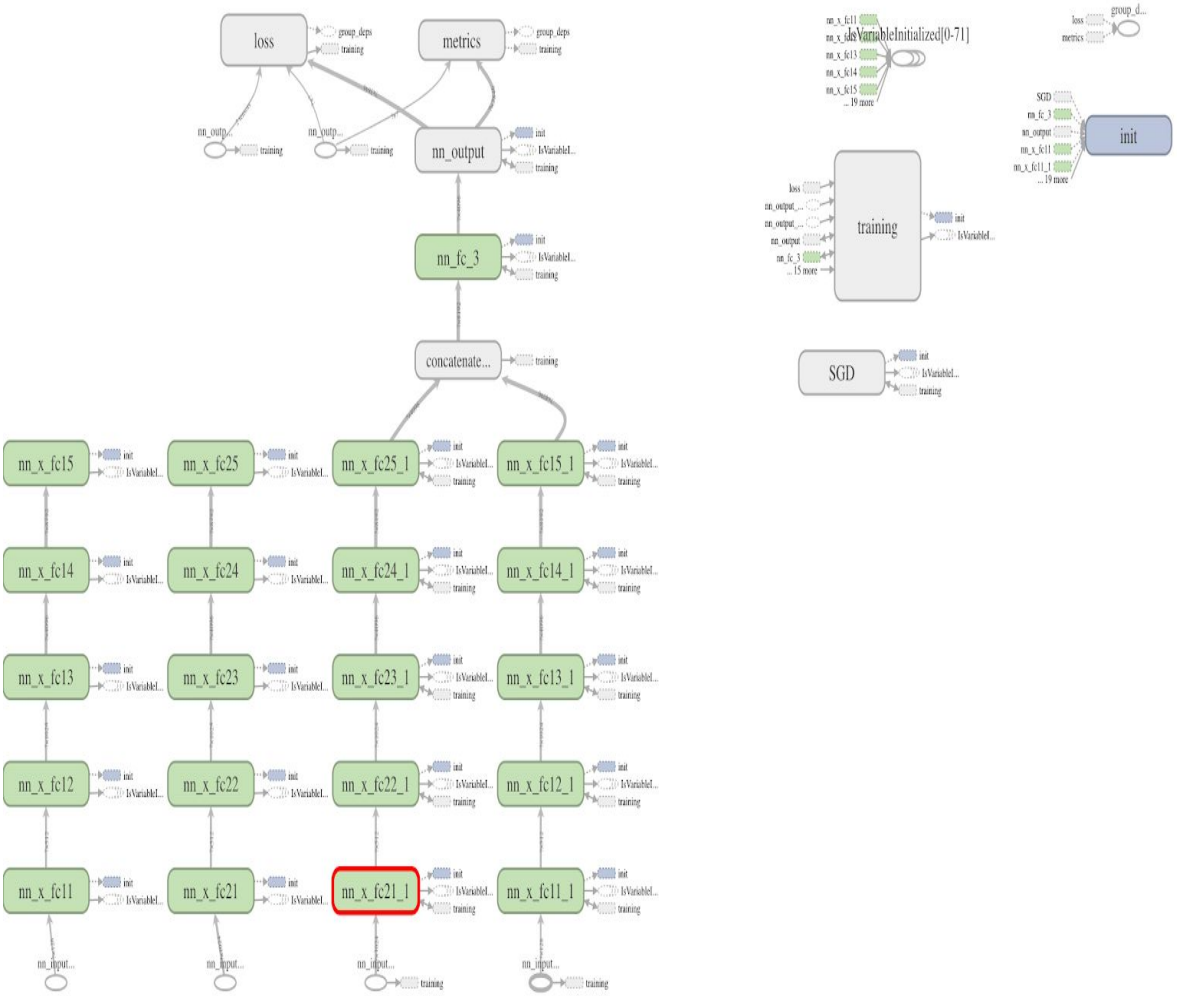


Figure 40: Above are system memory (Keras) experiments done in wandb.

Multi-Bidirectional LSTM

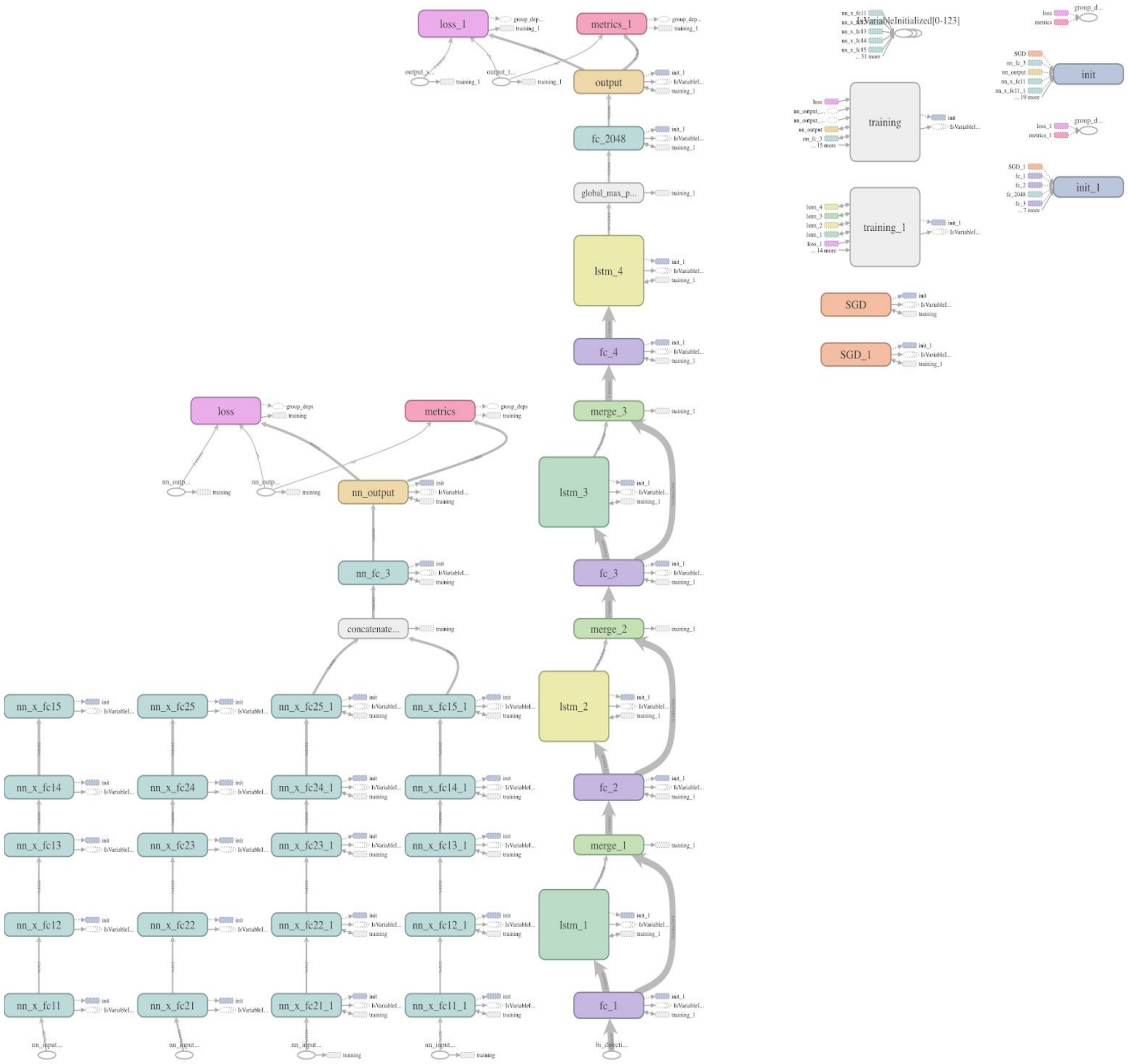


Figure 41: Above is a Bi-LSTM generated through tensorboard.

STREAM LSTM

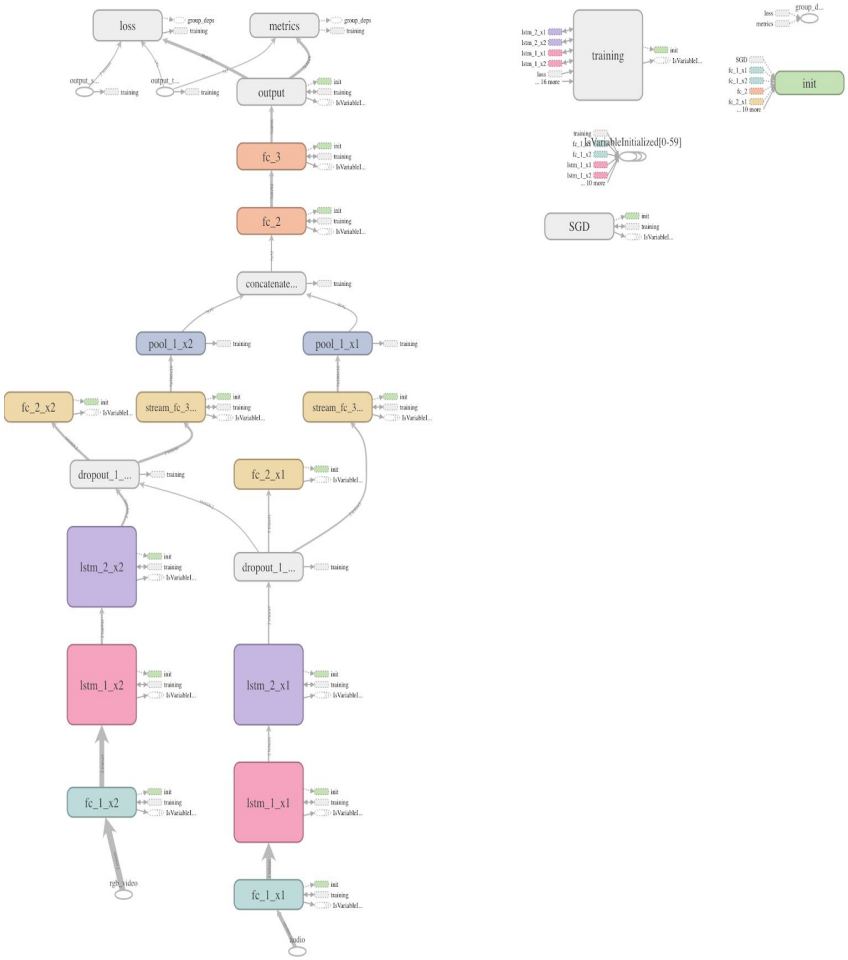


Figure 42: Above is a stream lstm generated through tensorboard

Neural Net LSTM Stream

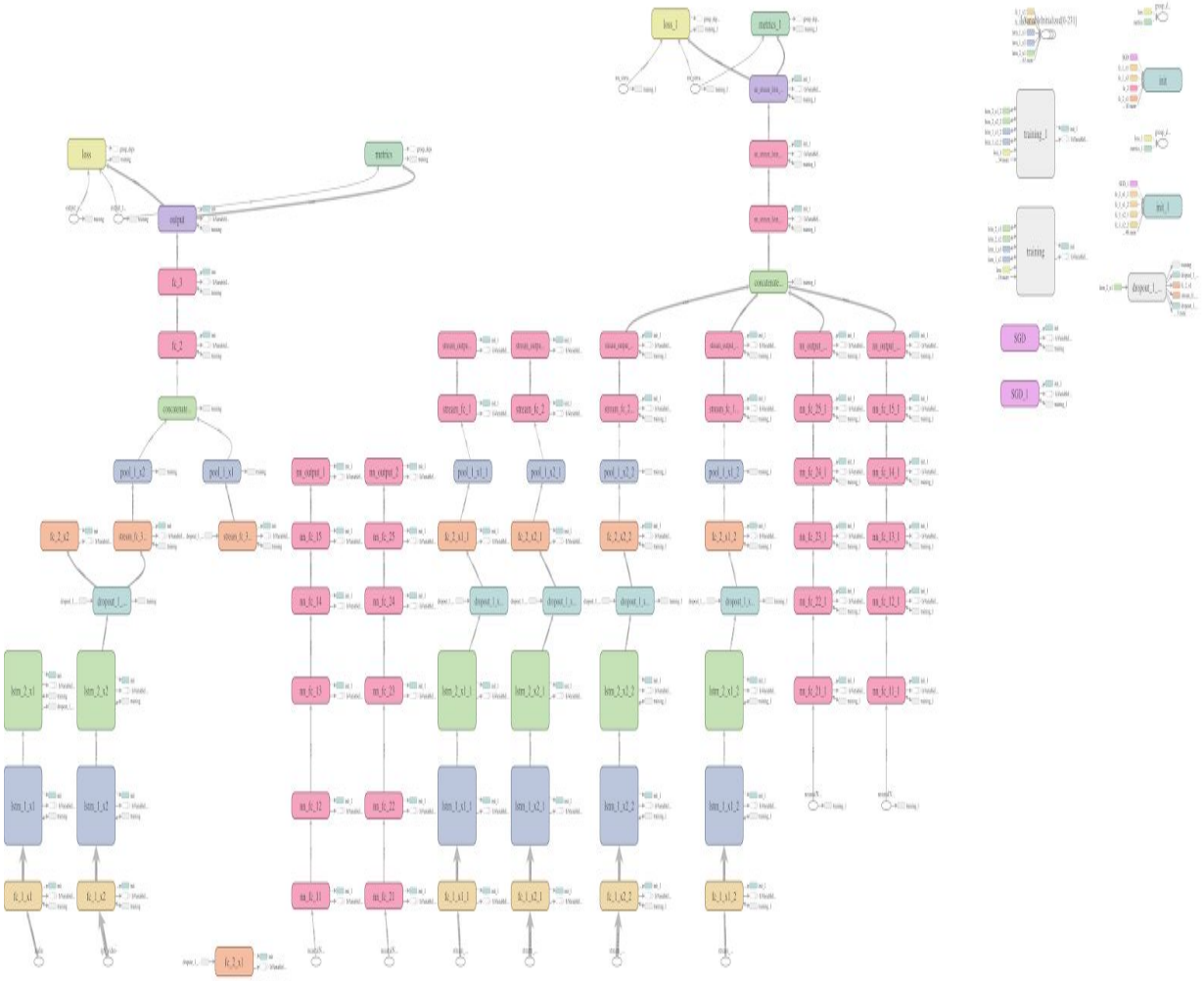


Figure 43: Above is a stream lstm generated through tensorboard