Classifying Video Labels from Youtube

Roberto Chavez

Advisor: Ram Akella

Second Reader: Roberto Manduchi

Thesis submitted for the degree of

Bachelor of Science in Computer Engineering

Computer Engineering Department

University of California, Santa Cruz

Contents

Abstract

The proposal for my thesis is to develop a classification deep learning algorithm accurately assigning video-level labels using the new and improved YT-8M V2 dataset from Google (Youtube) [1]. The YT-8M V2 dataset contains 8 million video URLs, 0.5 million hours of video stream, 1.9 billion frame features with audio & rgb, 3846 classes, and 1.8 average labels/video. All of which have been encoded (compressed) with a PCA [1, 7], later had the hidden representation of each video frame extracted prior to the classification layer. What was once a regular frame from a youtube video are now aggregated with weight parameters [2, 9]. If the aggregated data were to be displayed right next to a video frame from youtube, they would differ by a lot. The end result of aggregated data would look like a similarity matrix but messier. Google encoded the frame to make the content easy to download since 0.5 million hours of video stream is actually a petabyte long [1]. We went from 1 petabyte of video frame down to 1.5 terabyte hidden representations of a frame, saving a lot of space for data scientist to download. The tradeoff using compression loses content from the original frame. This is where I come in. In order to teach a computer to label a genre after watching a youtube video, I need to reconstruct the data using decoding techniques then encode again to: (1) fit the label size vector toward the end of my pipeline using feature engineering, (2)

provide more parameters to increase the gap of compressed info and (3) sequential data has not been aggregated yet. The data will be found on Google's A.I blogspot [1] or linked on my Github. This dataset contains both temporal and spatial content. Temporal is a sequential set of data to encode memorization problems like gdp predictions, or in this case audio-classification and a series of video frames. Spatial, the only features compressed, are high dimensional content like pixels inside a picture to classify objects in computer vision [3]. We are less concerned with the problem of computer vision itself but more concerned with the adapting class of machine learning known as deep neural nets to the problem of computer vision. There are going to be three factors to consider which algorithm is most successful: highest performance, least training time, and hardware usage (CPU, GPU, & Memory). All of which are important if you wish to deploy an algorithm into production after many hours of researching. I'll be coding my algorithms both in PyTorch and Keras to see which one I use for research and or production in the industry. If you developed an ML algorithm with excessive weight decisions, equivalent to a human overthinking, the design will be fine for research assuming the program doesn't crash. It'll be worthless from an industry standpoint, using an excessive amount of gpu usage to only lose more money with aws servers taking your algorithm into production. This is a challenging task because we need to manage the weight decisions for our algorithm in order to save memory and have a quick response time to inference. A great example would be the success of alphago, a computer from google who defeated the best go (weiqi) player, lee sedol, back in 2016. [42]. The bot was trained for a couple of months with 176 gpu's. Whereas it’s successor, alphago-zero, was only trained for 40 days with 4 gpu’s and no human interaction (no data) the following year [43]. Alphago-zero beat Alphago 100 times in a row. Alphago-zero is cheaper, faster and smarter. This is an example of deep learning improving immensely from a one year gap and the technology is only going to continue mapping out patterns in data, improving intelligence outside of human domain. My project is no different. The weight parameters of my algorithm will dictate the outcome of the performance. If my architecture has too many weight decisions from temporal compared to spatial, we will lose picture data in the optimization process (updating the weight/parameters to learn). The same event would occur losing audio data if we gave an excessive amount of weight values for spatial. Both classification techniques need to have a balanced relationship with their weight decisions, equal efforts to preserve information throughout the algorithm and workload with hardware usage. I will propose a couple of algorithms from both techniques, later concatenate both outputs to approximate what the computer is thinking after observing reconstructed audio & pixel data. Training all of the data is computationally expensive so I would need an excessive use of the GPU and CPU. I bought $100 worth of GPU/CPU cloud computation from "floydhub.com", later switch to “paperspace.com” for $400 (explained under work tools). In this paper I will discuss every model tested and see which one had the most success, and also the worst.

Machine Learning Goals:

(1) Find a problem to solve (2) gather the data (3) clean & transform the data to run on gpu's, (4) reconstruct the data using decoding techniques or feature engineering, (5) choose machine learning algorithm, (6) tune parameters (7) run experiments to measure loss, accuracy, hardware usages, (8) choose which algorithm would be useful for production services on a gpu cloud.

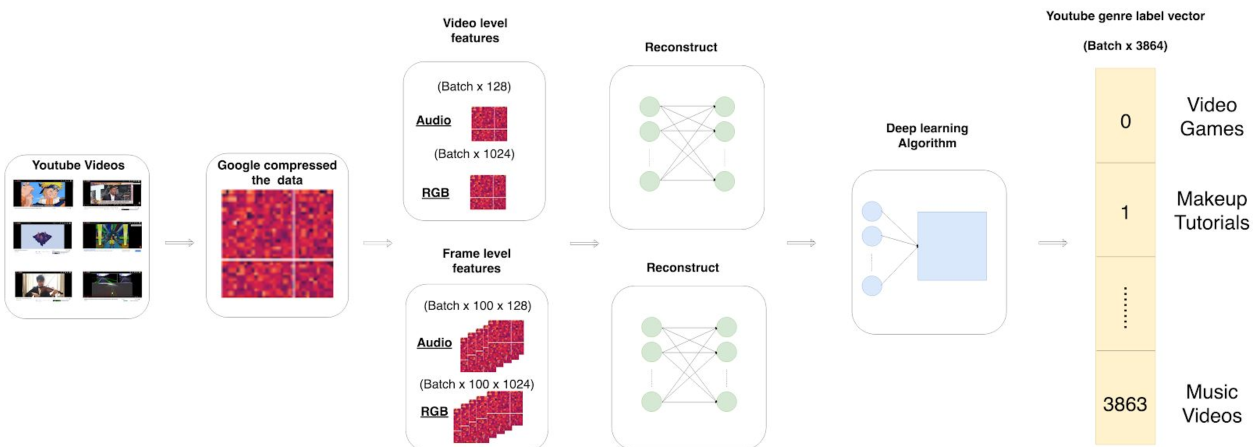


Figure 1: Diagram above is a pipeline of the project. I created 4 deep learning algorithms, coded 8: 4 in keras and another in pytorch. The most efficient algorithm is **Figure 2**. Diagram was drawn through draw.io

ACKNOWLEDGEMENTS

I would like to acknowledge Dimitris Achlioptas and Roberto Manduchi for giving me the opportunity to do my thesis in Deep Learning. Google for distributing their Youtube-8M dataset. Another thanks from Hugo Larochelle for teaching his online Neural Network course. 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.

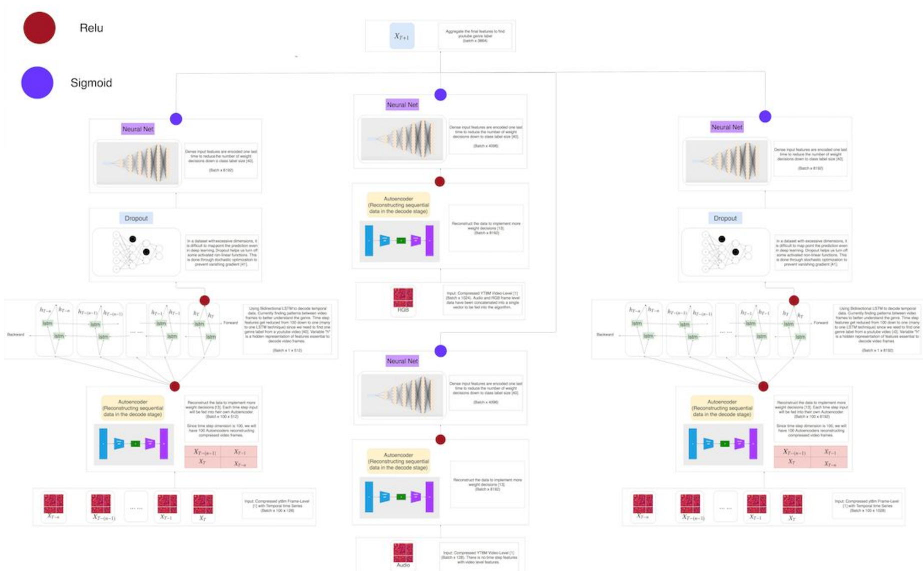


Figure 2: Above is a diagram I made through draw.io of a NN Stream LSTM. My model is too big to display I'll make sure to make them available on my github.

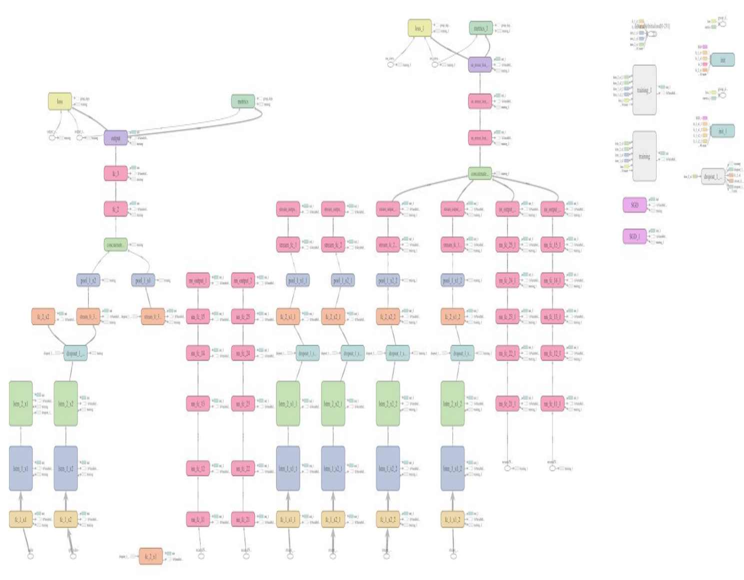


Figure 3: Above is a tensorboard diagram I created for my final algorithm. It contains a combination of Neural Networks for spatial content and an LSTM (Long Short term

memory) to compute sequential temporal patterns. More info under methods and see the image closer under my github portfolio.

WORK TOOLS USED FOR THE PROJECT

AI FRAMEWORKS

Initially when I proposed this project I wanted to code in tensorflow since it's an open source AI framework by Google. After coding the framework for the first month of the project I discovered it's constraints with symbolic programming [36] and horrid syntax. At least with tensorflow, symbolic programming is uncommon in python and your code performs computation at the last line instead of every line (imperative programming). Computation performing at the end of your code can lead to frustration when debugging since you cannot witness the cause and effect behind each line of code. The upside with symbolic is reusable memory to reference future task (generate computational graph). This is really important to deploy your machine learning algorithm into production using AWS servers. Computational graphs can be used to generate a tensorboard or setup manually with wandb. Tensorboard is a tool to visualize the structure of your algorithm. Wandb helps us generate real time results for: training, loss, accuracy, gpu/cpu usage and configurations to design a parallel coordinate (more info on experiment results) [27]. When wandb.init() is called from 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 so it'll be an interesting showdown comparing 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 framework built on top of multiple frameworks 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 in. There are two ways to build a model from their api, sequential and functional. Sequential allows 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 biggest downside I hear about keras is preventing the coder to control which variables can be utilize for the GPU or CPU. Two very important hardware specs mention later in this report.

PYTORCH

Most AI researchers in the past year have been coding PyTorch for it's dynamic imperative flexibility. The modern GPU-accelerated deep learning framework was founded by Soumith Chintala, an AI researcher at facebook who created the framework after his horrific experience coding in torch (lua) to compete in the coco detection challenge [35]. Their network was complex, it had multiple subsystems it was not always differentiable, resulting 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 task like rendering graphics 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 admitted 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.

FLOYDHUB

Since I have a computer that doesn't have a Graphics Processing Unit (GPU) to optimize my algorithms, I invested in a hundred dollars worth of compute credits from a website called floydhub. Their an online gpu credit service allows you run deep learning models. To be honest it was a poor experience running their site. The platform crashed everytime 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 nice. I then decided to invest in four hundred dollars of compute credits. Their compute service provided an NVIDIA 8gb GPU, 32gb of ram and 2 terabytes of online storage. In order to run my code into paperspace, you'll need to download my latest git commit (github) and upload the content onto your own 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 to classify images & video frames 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 become the next “moonshot”. A neural net is a machine learning algorithm inspired by the human brain to analyze data with a logic structure like a human. It’s existence has been around since the 1960’s but 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. Each weight parameter from each algorithm are aggregated with incoming input and 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 signal and negative vice versa. This is where training comes in, In the beginning before training, 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 in order to activate the correct neuron, sending a signal to the other neurons to 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 have a higher perception what the image looks like based on shapes and patterns of lines. Patterns 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 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 dimension 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. This can also be done with raw waveforms, later reconstructed to have a higher resolution in 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.

> Train: A portion of data 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. In order to decrease the loss, we need to send the difference into the the optimization process, an attempt to update our parameters (adam, stochastic gradient) for our next training cycle.

For example if a computer was given 95 images of a dog and 5 for a cat to train, It'll be difficult for the computer to tell what a cat looks like if given a new image (test) of a cat.

> Test: After the algorithm updated it's 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, your given the opportunity to repeatedly (epochs) look at old material (training data), later taking the exam (test data) with no permission from your professor to correct any mistakes.

This is known as 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 provide more cat pictures? In unsupervised learning this is a good time to use clustering techniques to mimic input data, in this case the minority labels, to generalize an imbalance distribution between labels. 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 is used to uncover input patterns, later help differentiating between categories (e.g. youtube genre labels). Before deep learning became a trend to extract heavy features, there was research in hand-craft representation for action recognition [18, 19]. The technique is based on local histograms of frames and motion gradients that have been extracted. 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. Training is used to update the weight parameters in order 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 are not updating during the process to see how the algorithm is applied to data in the real world it hasn't seen before. If the algorithm is **overfitting** from testing, the parameters are excessively added giving 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 using convolutional neural net (cnn) with short term temporal fusion of a pretrained 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.
Filter are used to aggregate pixels by sharing weight decisions, multiplying activated signal with the rest of the neurons in their respected layer to preserve computation compared to a single neural net. The output would increase the rgb depth channel representing extract features, another dimension. Any neighby pixels being aggregated with similar colors will be reduced using pooling techniques. Saving compute power on repetitive clorations. For example a picture of a stop sign is either red or white, assuming

This outcome capture short term info, making it difficult to compute yt8m since each video is encoded a hundred seconds long [1]. Another reason I can't implement their algorithm onto y8tm because it's compressed [1]. We may need is a logistic regression or a neural network to encode the spatial features or a combination with the autoencoder: a neural network representation to reconstruct data. [21].

Lot of useful paper essential for my project but lets transition into temporal related articles to encode long sequential youtube frames. There are two methods solving this issue. One way is to utilize conditional sequential memory like recurrent neural network [22]. Models like recurrent neural network started to have a prominent reputation to classify sequential data. Temporal content are stored inside hidden features to remember content from the past. An example would be reciting the alphabet backwards: most humans would have difficulty compared to a computer. One of the drawbacks of RNN is the limitations of weight decisions computed. RNN has the tendency to come across vanishing gradient, processing an excessive amount of computation: a.k.a weight decisions improve during training [34]. If I trained the model, I'd expect poor performances compared to LSTM. A model inspired by RNN I plan on discussing during implementation. 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. The only problem with the existing poolings mentioned, they are not trainable. More issues I hope to address under the method section.

Identity Mapping

If the data from yt8m becomes too complex 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 model initially sounds promising since the term “deep” came from adding more parameters than any other machine learning models. More expressiveness, better generalization. However most cases adding an excessive amount of parameters can cause two things: overfitting (already discussed) and vanishing gradient.

> Vanishing gradient crashes the system when we have an excessive set of parameters during training regardless the size of the training data. The accuracy will have low performance.

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 important to have deal with less parameters to prevent vanishing gradient while adding more to generalize. Microsoft use the technique to encode spatial data which is already done for us, but why not apply the same technique for sequential data? More info under methods implemented.

ANALYZING THE DATASET

There were two important task Google researchers [1] developed with the video dataset in a large scale.

1. Scaling time-consuming videos to annotate manual images. This was resolved by identifying 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 verticals popular on youtube were considered relevant knowledge.
2. Scaling videos are computationally expensive to process. Initially dealing with 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. Features were extracted 1 frame per second from 1.9 billion videos and was able to compressed petabyte content down to 1.5 TB.

In this section I will discuss Philipp Schmidt open sourcing his yt8m code from the kaggle community [46] to summarize the upcoming sections: **data processing, distribution between each class, and the probability of label occurrence**. I'm summarizing Phillip's kernel not for my benefit but to help readers understand the data visually before proceeding to my machine learning pipeline [46]. I saved all of the data graphs from philipp onto pickle objects [28], processing all of the data into would take weeks. I did perform feature engineering, making the data clean to feed into my algorithms.

Data Processing

Traditionally machine learning algorithms don't require a mass amount of content. yt8m contains 1.5 terabyte 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 respected CPU/GPU (paperspace: nvidia 1080) server 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]. This was the most time consuming step in the project since It took paperspace a week to process 1/4 of the dataset. After processing all of the data, I compressed a portion of the content into "pickle objects" for a quick demo since github has a limitation on storage [28]. If your interested in feeding in more data to train on, the code below 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 used to parse data from google's tensorflow database [1, 46].

What we have above are two for loops: first iterative looks at one file at a time while the second looks at one youtube video at a time. 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 access 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 it's distribution.

DISTRIBUTION OF CLASSES IN THE TRAINING SET

Cosmetics and other top labels needs to be closely reflected for unbiased distribution. Various estimation techniques based on finite sample is one way to derive the weights (parameters) to classify our classes with a less bias outcome.

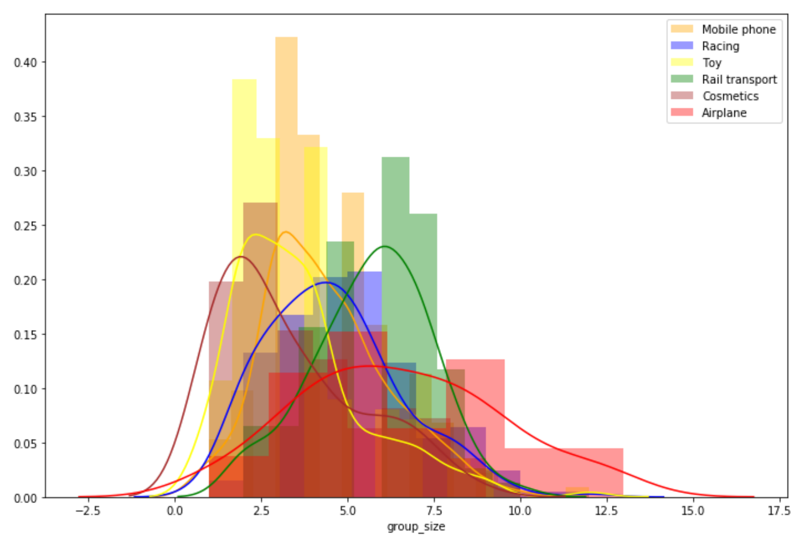


Figure 5: Above we have ourselves a label count distribution coded on jupyter notebook with [46]. On average each genre label has been grouped by 3-5 type of videos.

The group size distribution is quite arbitrary depending on the category. My computer couldn't process all 3846 unique labels on my graph so I only distributed 7 labels within the top 100 that are most uploaded on youtube. More than 7 unique colors represent labels on a graph would be confusing to look at. The graph was coded through a for loop from an array of labels with their respected index variable "labelArr". Same goes for the color array with character symbols. Both elements were computed through a distributed function.

PROBABILITY OF LABEL OCCURRENCE

At times a model can predict a label from a video similar to a genre we attended to classify. For example if I trained my model to classify a video labeled soccer game, the expected output would be a basketball game. Confusing the computer from the lack of parameters or lack in data distribution. Whereas a soccer ball and a football would be easily distinguishable if the data is clean and given the same amount of time to train. More data between two labels is one way to help increase the prediction.

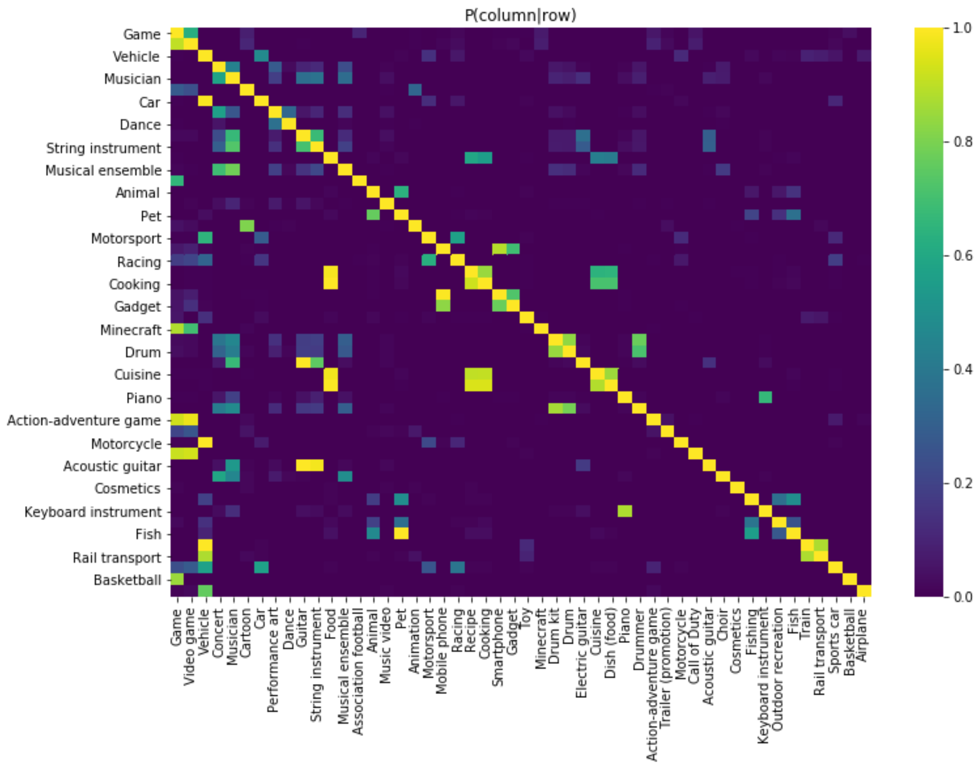


Figure 6: Above we have a similarity matrix representing the similarities between the labels since youtube videos have more than one genre [46]. An average video has three genre label but the data will only have one label each to make sure the least popular videos are trained.

Above is a similarity matrix coded of the estimated probability of label occurrence, given another label from sample data. This was coded in a double for loop: first loop recorded the sample data, second calculates the similarity measurement between labels. The label occurrence 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", "Foot-ball" 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? Either there is a bug mapping the labels to their actual names or this happens due to low sampling, since we don't use a lot of videos to estimate these probabilities. Overall the heatmap does show promising label occurrence that can be utilize for this thesis. If the performance for my algorithms is relatively low, I would need to compare more distributions on the least popular videos on youtube (real estate) and synthesis more data for them. Now that we analyzed the data sample it's important to observe the video and audio features and how can I create my model.

VIDEO LEVEL FEATURES

If you wish to make an easy machine learning algorithm to predict a genre, video-level features should be your first stop to analyze. There are no sequential data in video level data. Spatial features has already been compressed unlike temporal data so you will expect performance with algorithms differ in the next experiment section. The feature represents an entire video with rgb and audio encoded. Kinda like a thumbnail you see after searching a query on a youtube video.

I randomize which rbg labels should be graphed below in order to explore what needs to be regularized? What this tells us for RGB features, every data set with video label has been gathered properly that doesn't require difficult regularization.

> Regularization: Reduce the number of parameters to prevent overfitting: an excessive amount of parameters to only memorize content from training instead of generalizing a mapping pattern to predict test data. With neural nets we use dropout for regularization.

Less bias computation for each video label. 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.

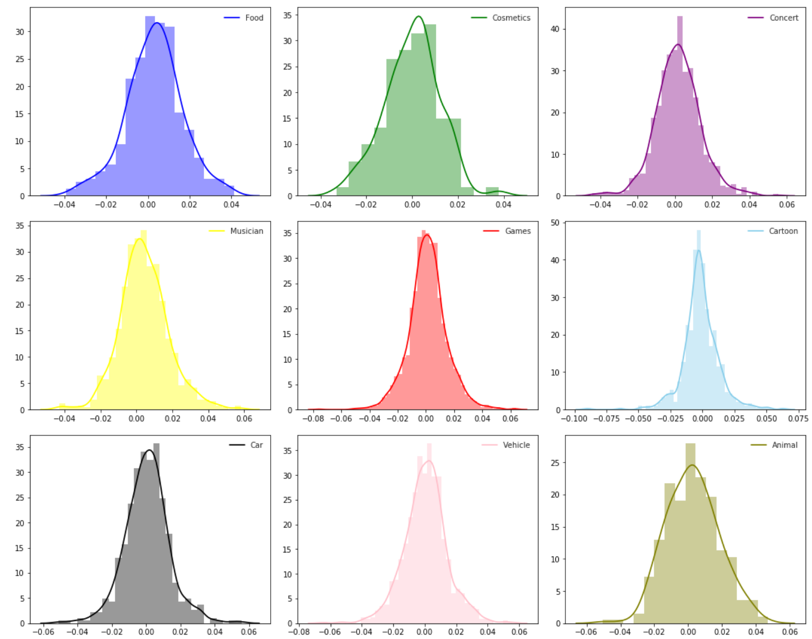


Figure 7: Above we have an equal distribution between each label since we're only training with one genre per video. I randomize which genre would appear above from 3864 videos, giving me an unbiased perspective between labels.

FRAME LEVEL FEATURES

We examine audio frame level features of a youtube video. The frame feature extracted from taking a sample each through a video [1, 6]. Since we're dealing with sequential data with frames, it's important to know the higher distribution in order to set a regulation to equalize the number of data per frame.

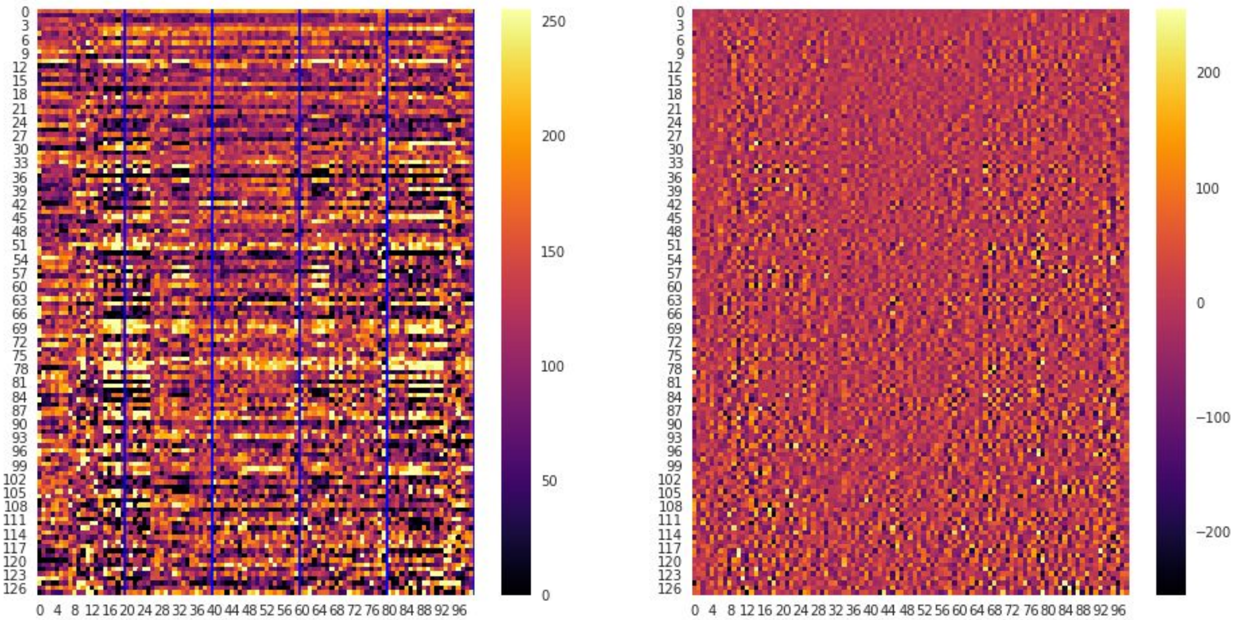


Figure 8: Above we have a 2 dimensional array of a frame containing features from a 100 second youtube video. Each column represents a second of a youtube video.

Above we have a two-dimensional array of one video frame. These data arrays represent a spectrogram for each row displaying frequency whereas each column is a timestep. The color represent the magnitude of the audio frequency. It seems like each video has a high pitch frequency magnitude in the beginning and end to a youtube video. This high pitch can be the source of information that can be fed into a temporal model that can account .   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 (Implemented)

I coded 8 different algorithms, 4 in each AI library (keras & pytorch): Neural net, Multi-bidirectional lstm, stream lstm, and a neural net concatenated with an stream lstm. I plan on discussing different properties used to create each algorithm. All the math will be discussed under Appendix A. One thing I like to point out with my proposed algorithms, the model with the highest accuracy doesn't make it superior to another. Every algorithm has their own success from previous papers and every outcome differs from each dataset. There's going to be three types of models for our architecture: video-level, unsupervised, and frame-level. Models include:

- Autoencoder (Unsupervised deep learning model): Model known for reconstructing compress data by generating new content in order to mimic the input. Except were given an input of encoded data from youtube. For audio alone were given a input size of 128 whereas the output class video labels is 3846. This means we need to expand the dimensions of the audio input with weight parameters.
- Deep Neural Multi-class label classifier (Video-level model): A neural net with binary logistic regression, multiple layers to classify a video label.
- Bidirectional LSTM (Frame-Level model): Two LSTM connected back to back to back.

- Stream LSTM (Frame-level model): This work was inspired from two previous models. LSTM and Deep Residual Nets [35].
- Fully Connected Neural Net + Stream LSTM (Video + Frame Level model): Combine the softmax between the two to find our final model. Our most prominent models combine.

Training a model requires data to help the model update their parameters similar to studying for an exam. This process is repeated multiple times until the number of training batches and epoc iterations are finished. After explaining each model presented in this project, I will explain how each one was created through keras or pytorch.

AUTOENCODERS

Since the data is compressed, we need some form of a reconstructor with a neural network architecture: an autoencoder was the ideal choice [13]. Every single algorithm below is going to an autoencoder, to later decode our compressed data. The size of my input, 128 for audio and 1028 for rgb, is unusually smaller than the output youtube size 3846. In the case of small input size, mentioned under related work, we would increase the size of the input with a reconstruction. We do matrix multiplication between the input and the weight parameters to decode more info out of the input. This gives the input more room to leverage, narrowing the output label with more possibilities to consider from the number of parameters added. Usually an autoencoder has two properties: an encoder and decoder. The encoder is compressed data, reconstructed through the decoder in order to mimic the input with less features. Since google compressed the data [1], the best decision now is to reconstruct the input using the autoencoders decoder. This method will be applied to every algorithm. For both keras and pytorch, we increase & decrease the number of hidden units and make sure you have a binary cross entropy loss inside the optimization process.

Deep Neural Net (MULTI-CLASS BINARY CLASSIFIER)

The model has video level rgb and audio features fed into a one-vs-all binary logistic regression. Each label trained on video-level features. Binary logistic is another word for nonlinear function mapping data with a large amount of features like a picture. These weight are multiplied with youtube frames, aggregate important features after training, later output a probability number between 0-1 using activated functions. Base on recent studies, relu is the most optimal function to prevent vanishing gradient [30]. With optimization, weights are derived in calculus, changing the values through stochastic gradient. Once I’m done creating hidden layers, 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 to calculate the lost .

CODED IN KERAS

Using Keras’ dense api function (linear), we initiate a number of weight decisions to do element wise multiplication, aggregating the input. Inside the dense function has an argument called "activation", enabling you to pass the string "relu" to use the rectifier function. Once I finish coding the hidden dense layers, I concatenate two hidden layers (rgb & audio) into one using keras.layers.merge library which is also the final output. Pass in sigmoid as an activation for the final dense layer (output) and train the model using the fit model class.

CODED IN PYTORCH

Since PyTorch is imperative programming, I can code my model using object oriented programming. Creating a class to support the model architecture and a forward function to pass in input data, return an output from the model. I initialize my layers and activate operations inside a python class called “Neural_Net”. They are saved as private variables and are wrapped through nn.Sequential, a class from pytorch to add modules: layers and/or activated functions. Modules added in the order they were coded is 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 in order to output something meaningful. The output vector contains either a one or zero: one represent the classified label and the other should be zero for having the lowest approximation. Architecture is finish 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.

RNN vs LSTM

Recurrent Neural Network (RNN), similar to a regular neural network, is a model that can compute temporal sequential patterns. Their computationally expensive compared to spatial algorithm since their memory intensive on the hardware usage [22, 34]. We need a model that can preserve parameters, predict further outcomes in the future. Unlike RNN, LSTM has a cell state to control what parameters are relevant to preserve longer 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 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 how I develop my pipeline. We’ll see the performance gap in the experiment section. These gates are controlled by a sigmoid neural net layer. The sigmoid function inside a forget gate outputs a number between 0-1: 0 for being irrelevant to the video label and 1 the opposite. The content with the highest score to 1 gets to be preserved for the next cell state. We utilize this model to see a series of video frames, 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 needs to be approximated. The dimensions for the input will be the same like RNN and Keras has 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. This is useful when the 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 backwards "...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 own LSTM to concatenate a fully connected dense that will be sent to the final output. Back to the data, both rgb and audio from frame-level data will propagate into their own LSTM models. RGB has the dimensions of (batchx100x1024): 100 for 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 for number of frames from 100 seconds in an audio, and 128 for the number of audio features in one second. Once we have both features computed through their own LSTM, the output is concatenate 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 sequential data backwards and return hidden/output units from both models into a vector. The first half of the vector will have units from hidden & output to make the first lstm and the second half the second lstm. The second lstm with bidirectional flag will be set to true summing up to four lstm. From my experience stacking up 4 layers will crash the server for an excessive amount of parameters to compute.

Stream LSTM (IDENTITY MAPPING + LSTM)

I've decided to have 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 important modules in this design: a fully connected dense layer and an LSTM. 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 gradients: 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.

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 audio pattern for one time direction .The next set of lstm will decode the same data except returning backward sequence (go_backwards=True), decoding sequential data from the end of 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)
```

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 and cell states are already created inside the function. Each hidden & cell state are fed into the lstm pytorch model class

```
# 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))
```

Above is only a snippet of my algorithm. Full source on my github

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.

Neural Net concatenated with an 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 concatenate, meaning the probability between each label was averaged between both models. Both keras and pytorch both had their own concatenate/merge class with little difference.

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 an 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 frameworks is to analyze the relationship between the two, later decide which one I would use for research and/or production in the industry. I experimented more than once for each algorithm, later figure out what settings for the batch size, epochs, and learning rate are favorable.

> Batch size: A subset of a data set 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 fed into the machine learning model. If we had two epochs and three batches, the total number of iterations would be six. Thergo our parameters were updated six times.
>
> Learning rate: The rate how quick a model decides to abandon it's old parameters. If the rate is low, the parameters will be able to lower the lost but at a slow rate. If the rate is high, the parameters will update but

constantly 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 can I monitor my hardware usage? After discovering wandb in november, a startup that's went 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.

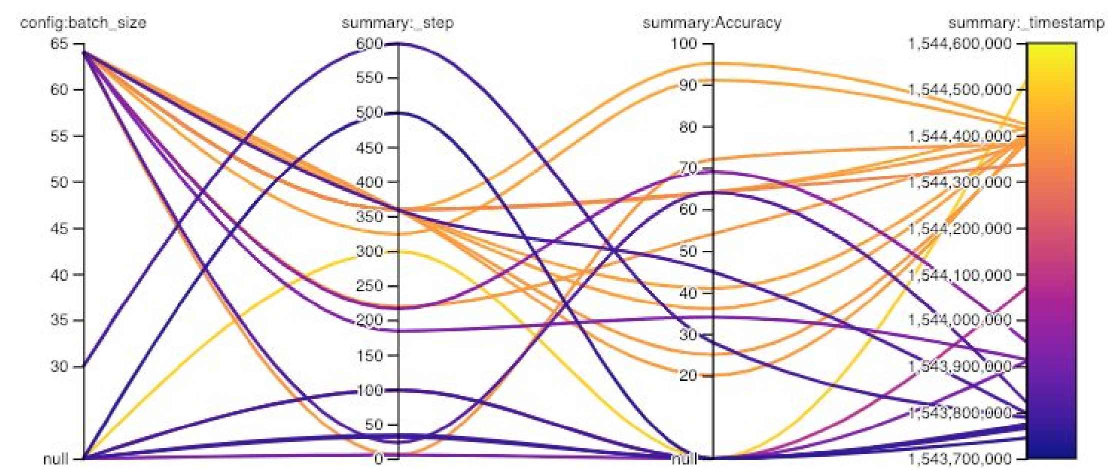


Figure 8 : Above is a parallel coordinate generated from wandb after training each experiment in pytorch. (View my wandb account to interact with the graph)

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 experiment 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 important graphs from my experiments: accuracy performance, number of iterations to train, and memory management.

Training (PyTorch)

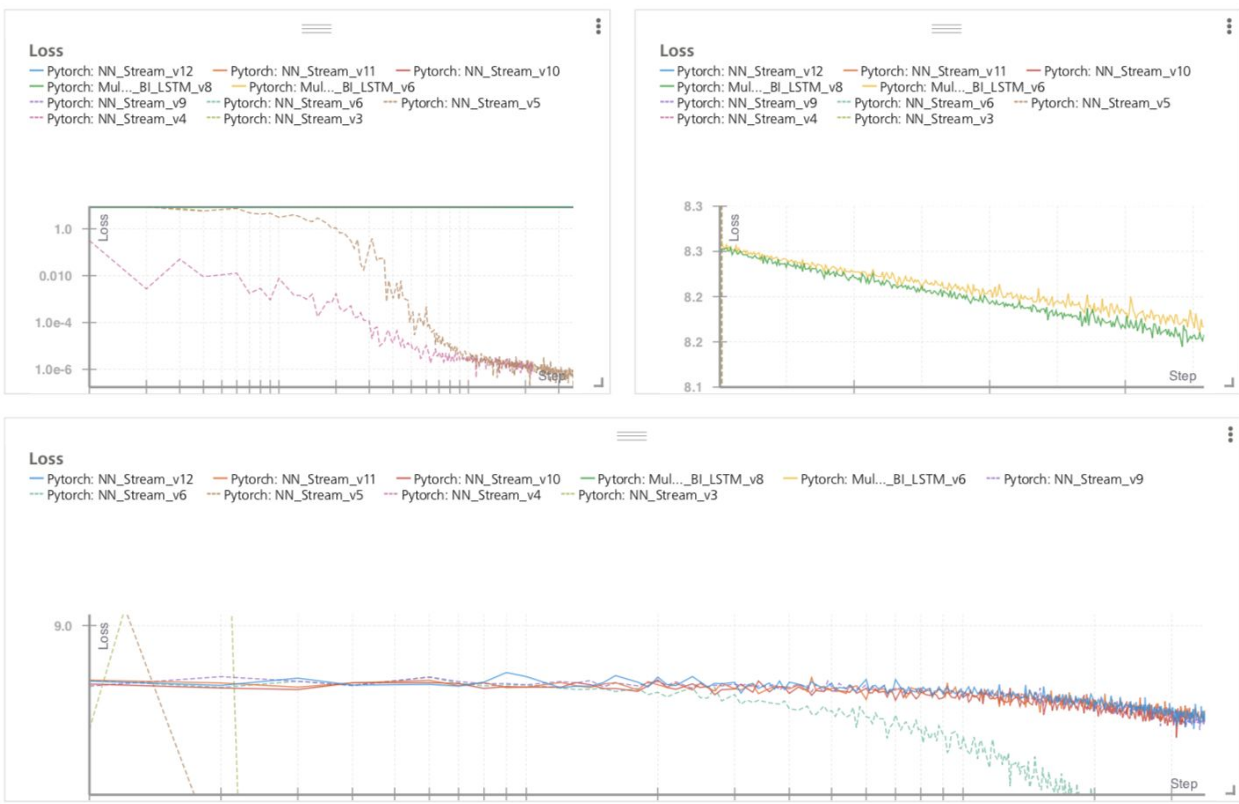


Figure 9: Training session for every pytorch model I experimented on wandb
(View my wandb account to interact with the graph)

Initially when you first start training your network, your lost function might have an immediate dip (above graph) since the parameters have been randomly initialized. The gap between the predicted output and the expected results is huge since random parameters 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. Looks like our model is improving. The lower the loss, the higher the chance each accuracy will have optimal performance.

Accuracy (PyTorch)

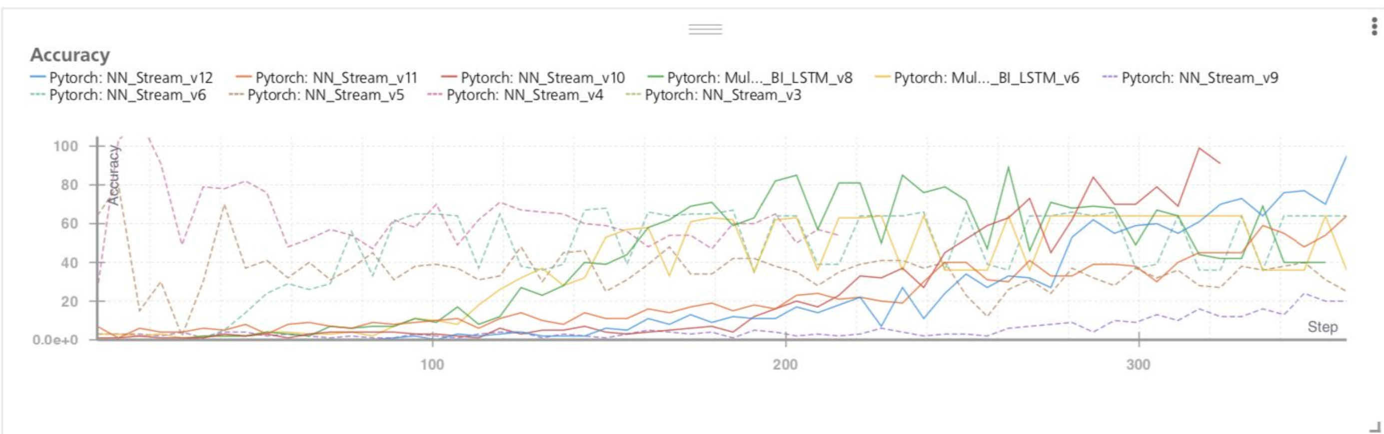


Figure 10: Accuracy session for every pytorch model I experimented on wandb
(View my wandb account to interact with the graph)

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 parameters 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 accuracy for each algorithm starts to become consistent after 150 epochs, the same moment of time the lost function had a consistent decay. The biggest factor to all of my designs were 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 abritraily 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)

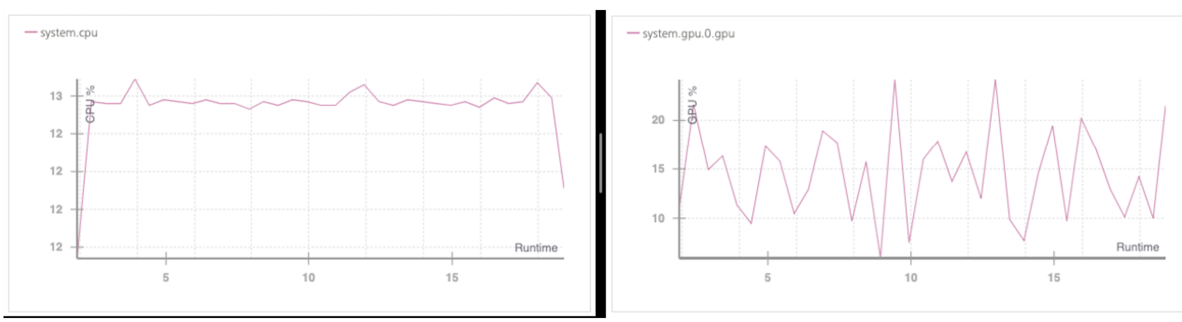


Figure 11: CPU/GPU session for a pytorch model I experimented on wandb
(View my wandb account to interact with the graph)

A neural net concatenated with an lstm stream manages its hardware usage efficiently. Running the algorithm quickly using a low usage on the cpu and gpu. Program would of crash if the hardware usage 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 be deployed into production for an api cloud at a very low cost. The amount of training was relatively low, making it cheap. The higher the cpu/gpu usage, the more expensive it cost.

Training (Keras)

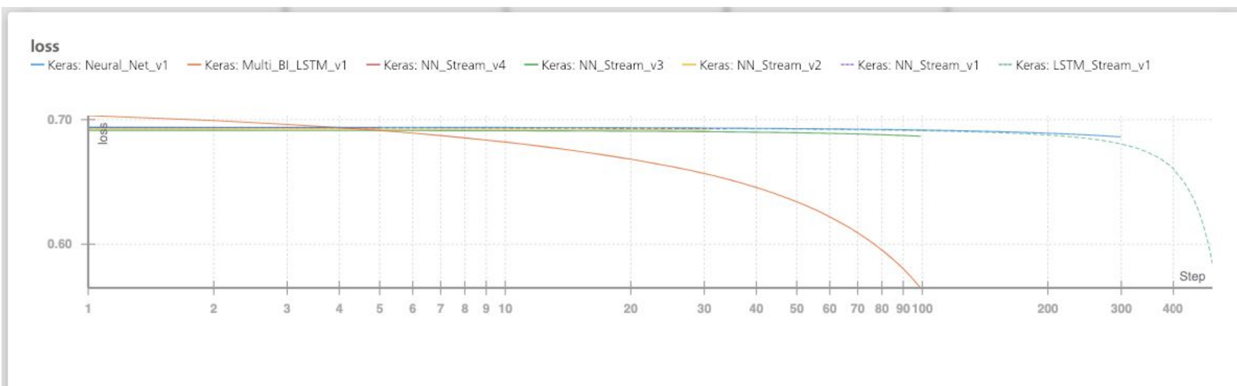


Figure 12: Training session for every keras model I experimented on wandb
(View my wandb account to interact with the graph)

Most of my models initially trained at 70%, making the difference between pytorch 1-9% results wider. Each model in keras took twice amount of time compared to pytorch to train. It's possible my current results are based 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 are the behavior between loss metrics. PyTorch had a noisy reaction while keras did not, I think keras has an api to clean 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)

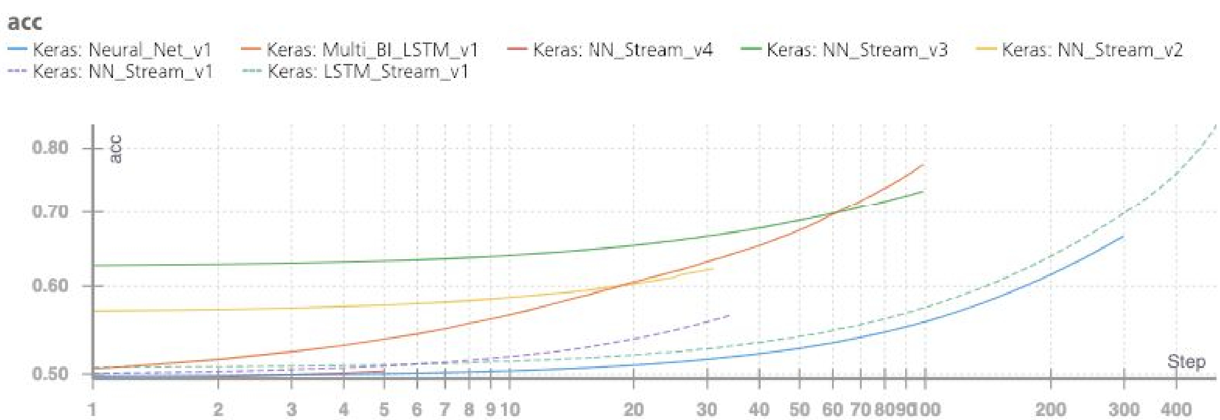


Figure 13: Accuracy session for every keras model I experimented on wandb (View my wandb account to interact with the graph)

The accuracy for most models in keras started at 0.50 (50%), or greater, 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 possibly crash, using an excessive amount of cpu/gpu. The metrics are not compulsive as pytorch in 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 a higher accuracy but the hardware would be an issue (mentioned later).

CPU and GPU usage (Keras)

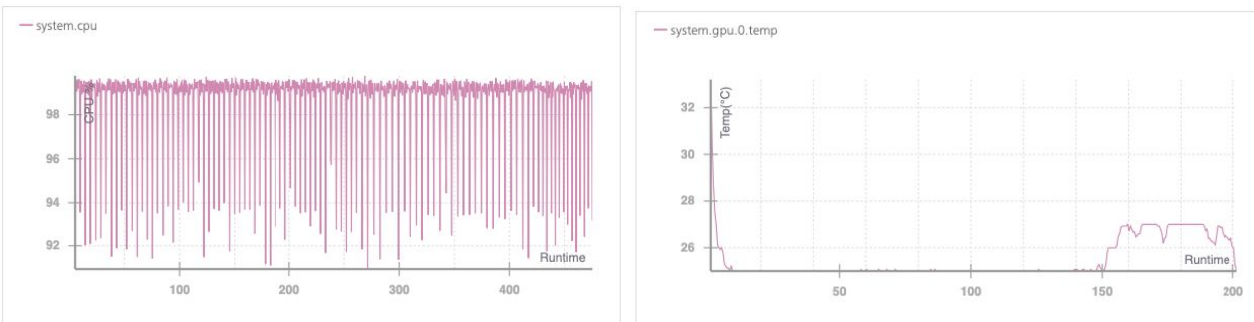


Figure 14: CPU/GPU session for a Keras model I experimented on wandb
(View my wandb account to interact with the graph)

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 an stream lstm with 94% cpu usage and the temperature for the gpu rose up to 27% during training with no memory usage. This could mean the algorithm was slowly beginning 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 even 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 15: CPU/GPU session for a Keras model I experimented on wandb. All of the experiments are available on appendix b section.
(View my wandb account to interact with the graph)

After experimenting with both frameworks, I think keras is easier to code but with less control to train between a gpu or cpu. 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 help me value my profit throughout the experience.

Concluding remarks and Future work

There are several ways this project can be extended. If the dataset were uncompressed from google researchers [1], we can use generative models to create new youtube videos [28]. This would give more possibilities in video game graphic research. 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 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 estates are not heavily 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 in order 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

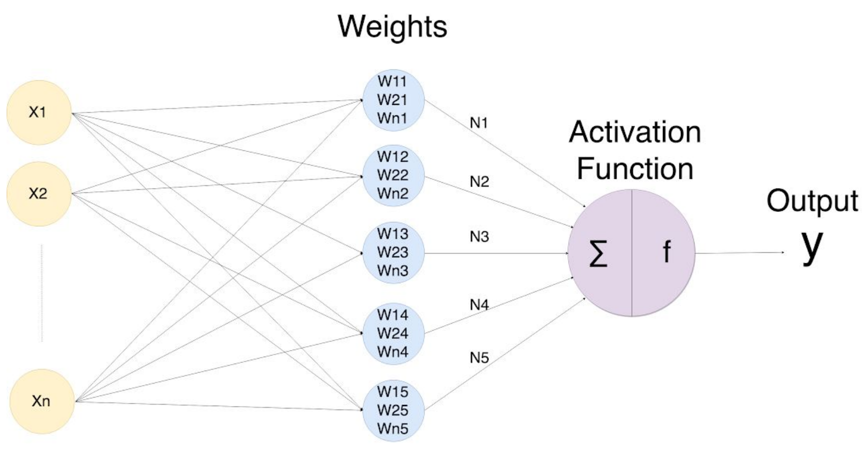
[45] Johnson, Khari. “Facebook launches Pytorch 1.0 with Integrations for Google Cloud, AWS, and Azure Machine Learning”. venturebeat.com.
https://venturebeat.com/2018/10/02/facebook-launches-pytorch-1-0-integrations-for-google-cloud-aws-and-azure-machine-learning/?fbclid=IwAR0ZbFcn9U-plAx5uiKEsbosACSTjvoNruQsJkesgRbbqSHYx67Mu2M7_YE

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

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 are 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 16: Above we have a neural net structure of our yt8m data ‘x’ being aggregated 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 inputs. You also see a table representing the aggregation between the input and the weight decisions. It’s a higher level intuition how computation is done in linear algebra (matrix multiplication). Yt8m video level data will be the input while the weighted sums ‘w’ multiply with their respected sums.

Scoring input of the model:
input * weight = guess

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

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

$$max \tag{3}$$

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

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

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

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

Figure 17: Above we have the formulas used for in a multi class neural net. Above was coded on a seperate latex file

Above is the predicted output from the rectifier layer unit. It’s one of the many activation 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 signal to send more information to the next depth of neurological comprehension.

Autoencoder

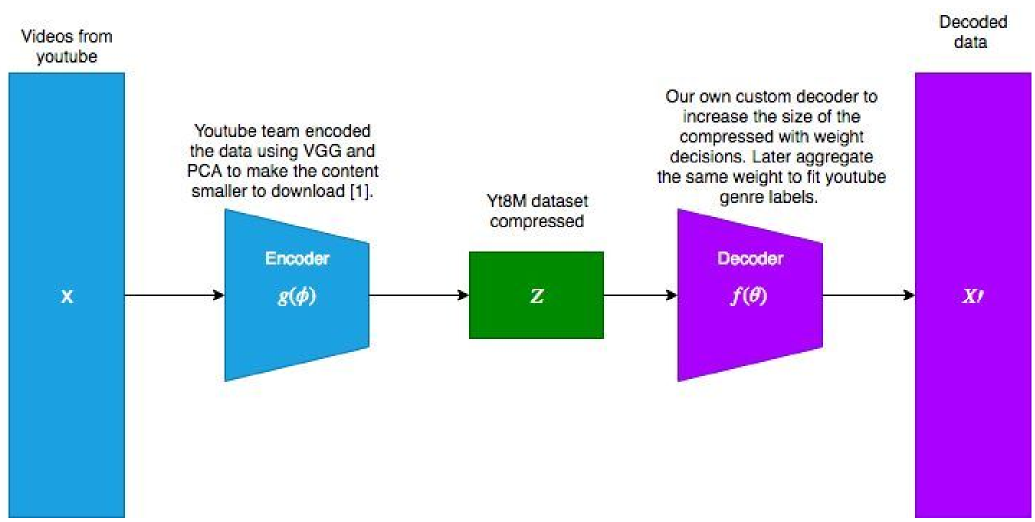


Figure 18: Above we have an autoencoder which is in the form of neural nets reconstructing the data.

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

Deep Neural Net

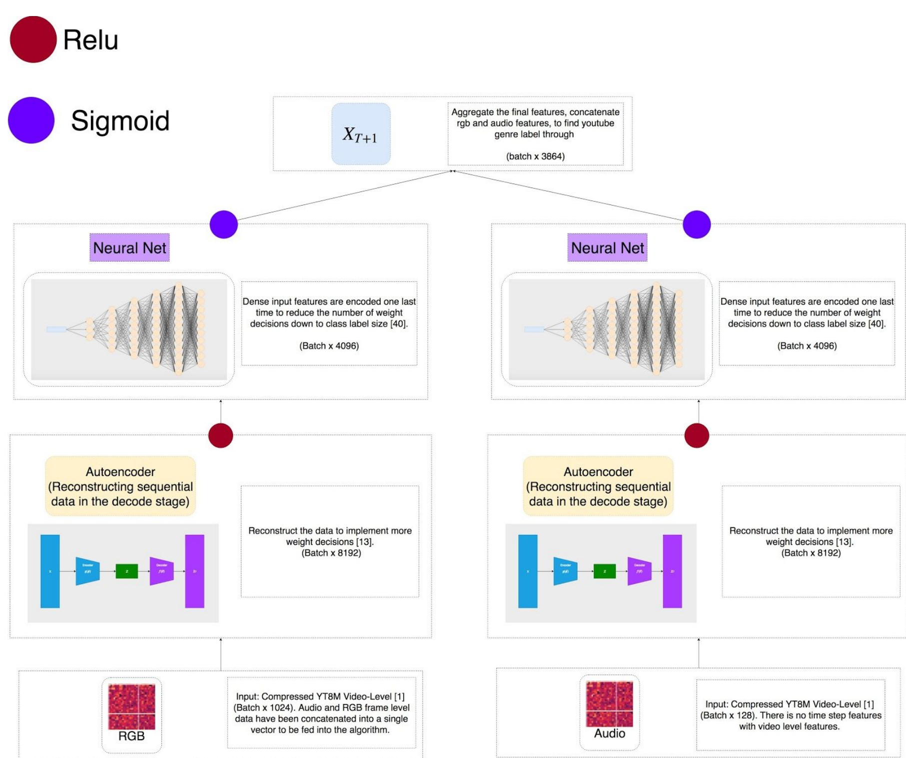


Figure 19: Above is a deep neural net used for the first series of experiments for easy design. Diagram was drawn using draw.io

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.

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

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

adjustment = error * 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.

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

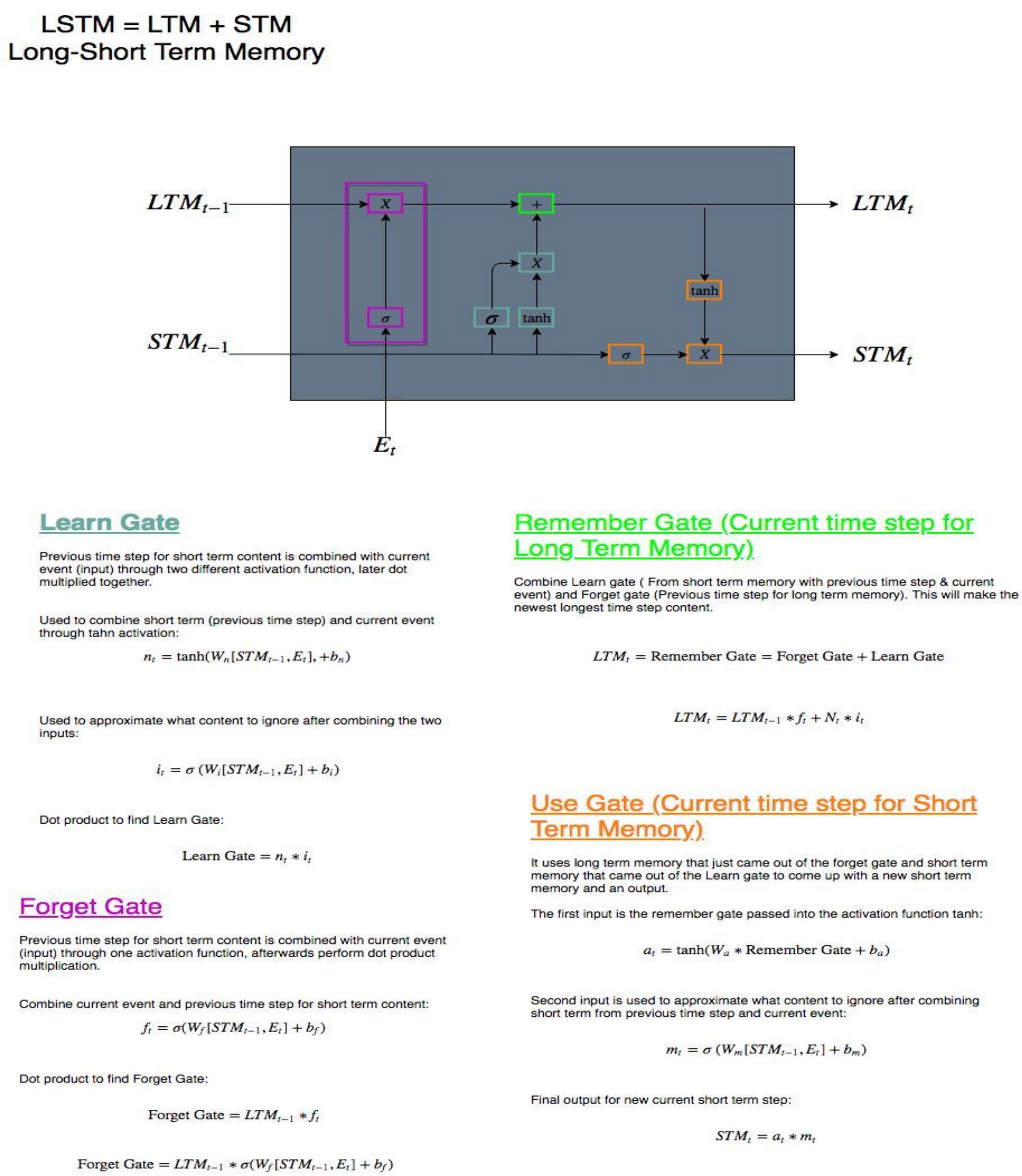
Figure 20: Above we have the optimization process used to update the weights. This was formatted on my private latex to screenshot.

Given two variables, error and weight, activation is a variable intervening the two. Activation is responsible to approximate the probability between labels in a given data set [33]. Change in weight affects activation and change in activation affects error. Variables for activate and weight are differentiable to effect the error. Below are the ideal function to activate signals.

1. Bounds: Limitation for our activated output needs to be in between (-inf, inf). For example, a sigmoid is only between 0 and 1 whereas a linear function (ax + b) limit is (-inf, inf). Any one axis is given permission to be inf so as long both aren't.
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): Making 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 complex 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 21: Above is a formula sheet I made of a LSTM architecture. We have the following: Learn, forget, remember, and use gate.

Multi-Bidirectional LSTM

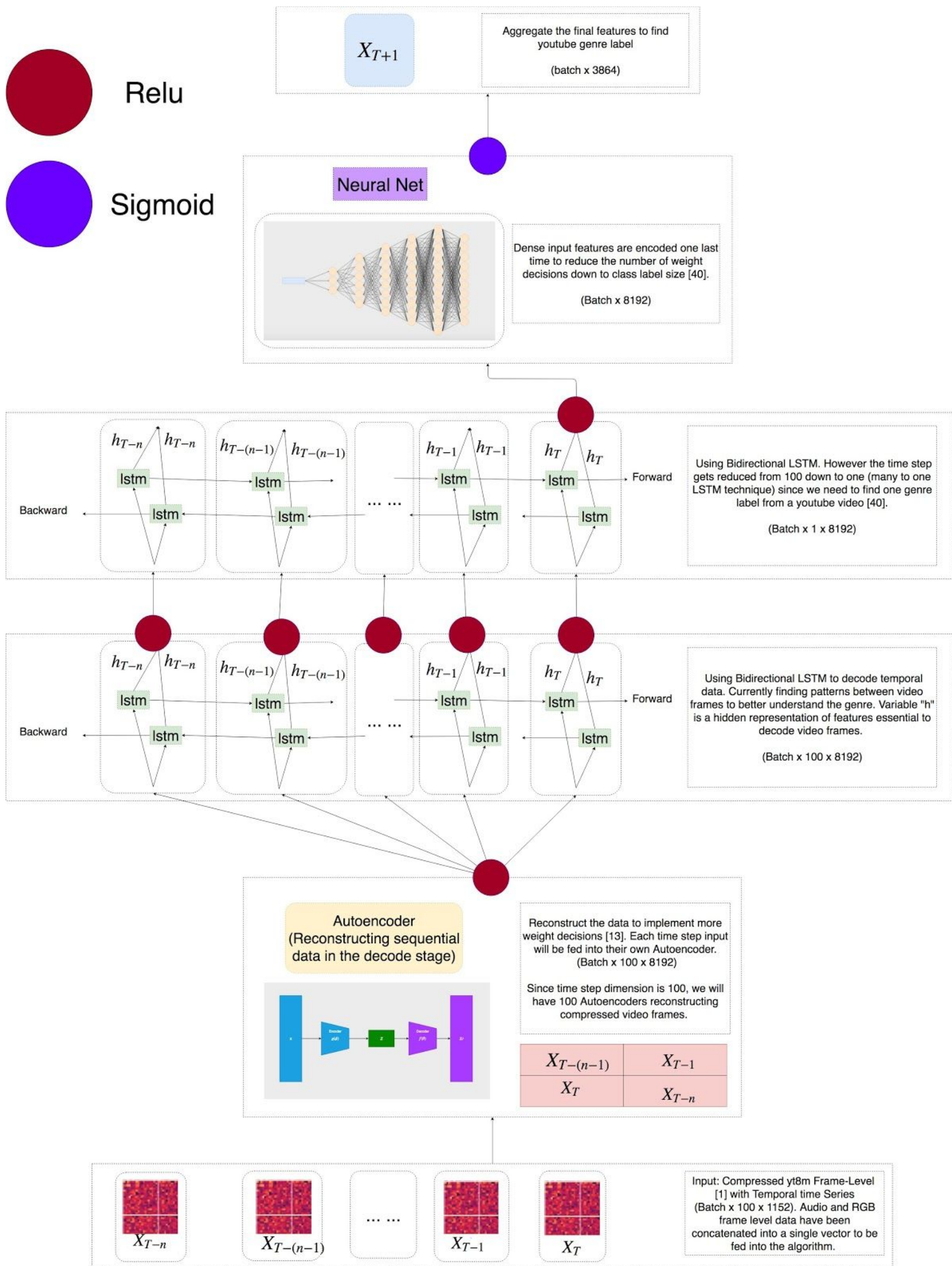


Figure 23: Above we have a bidirectional lstm created on draw.io

Stream LSTM

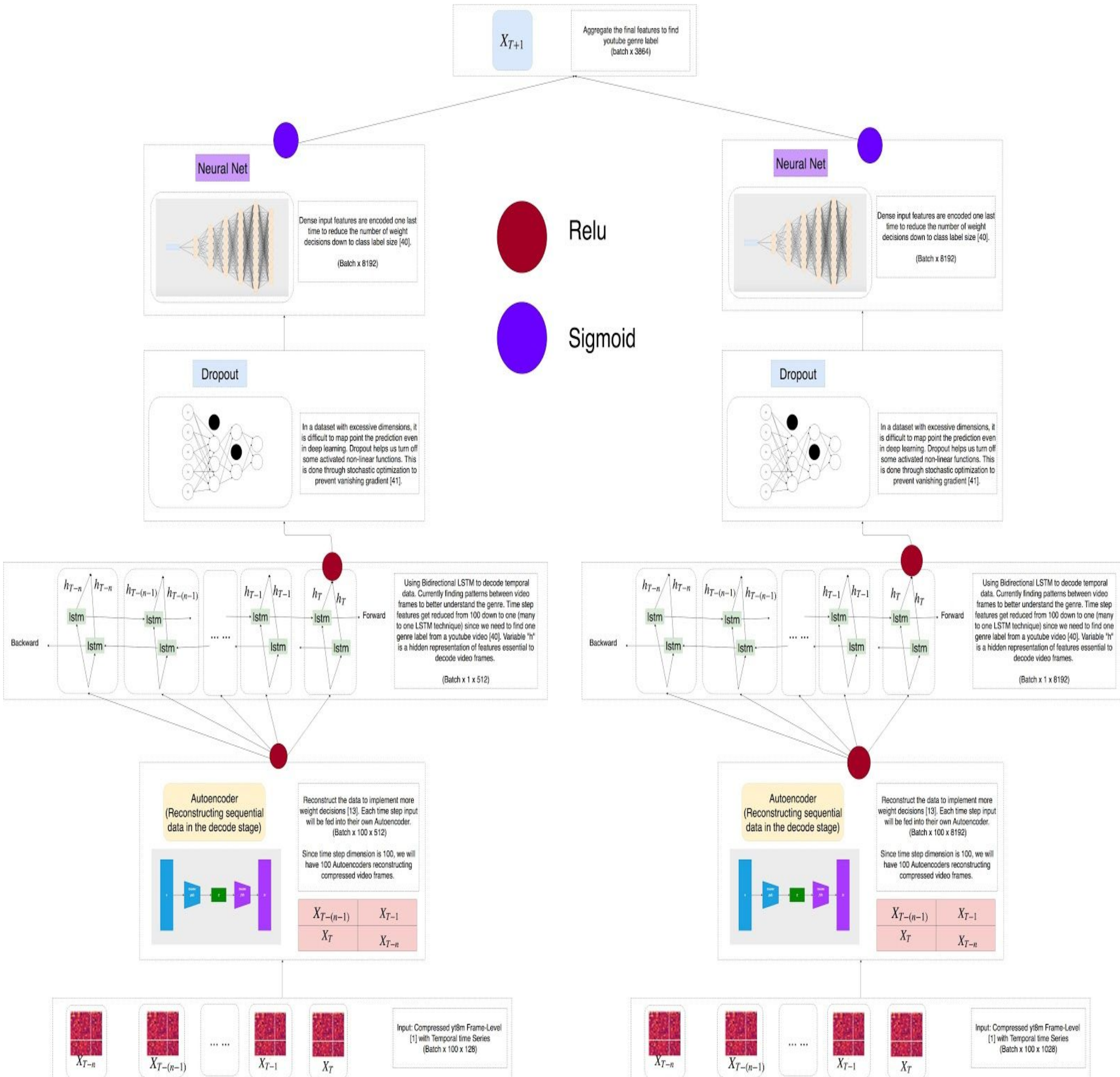


Figure 24: Above we have a stream lstm created on draw.io

Neural Net LSTM Stream

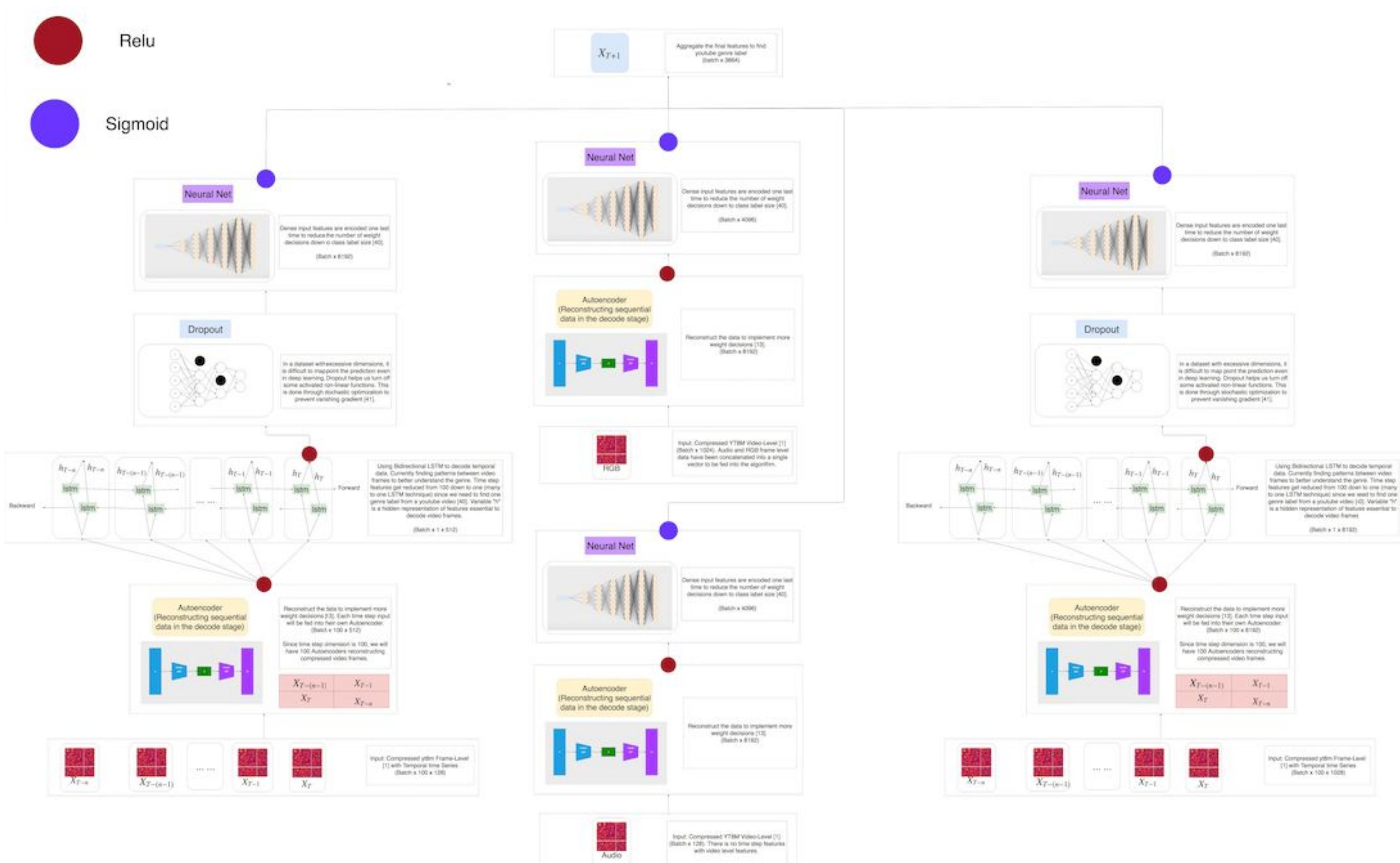


Figure 25: A Neural_Net_LSTM_Stream created on UML (draw.io). This is the final design for my algorithm and it's a lot to process all on one image. Some of the text is not visible so you will need to visit my github portfolio to see the entire diagram.

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.

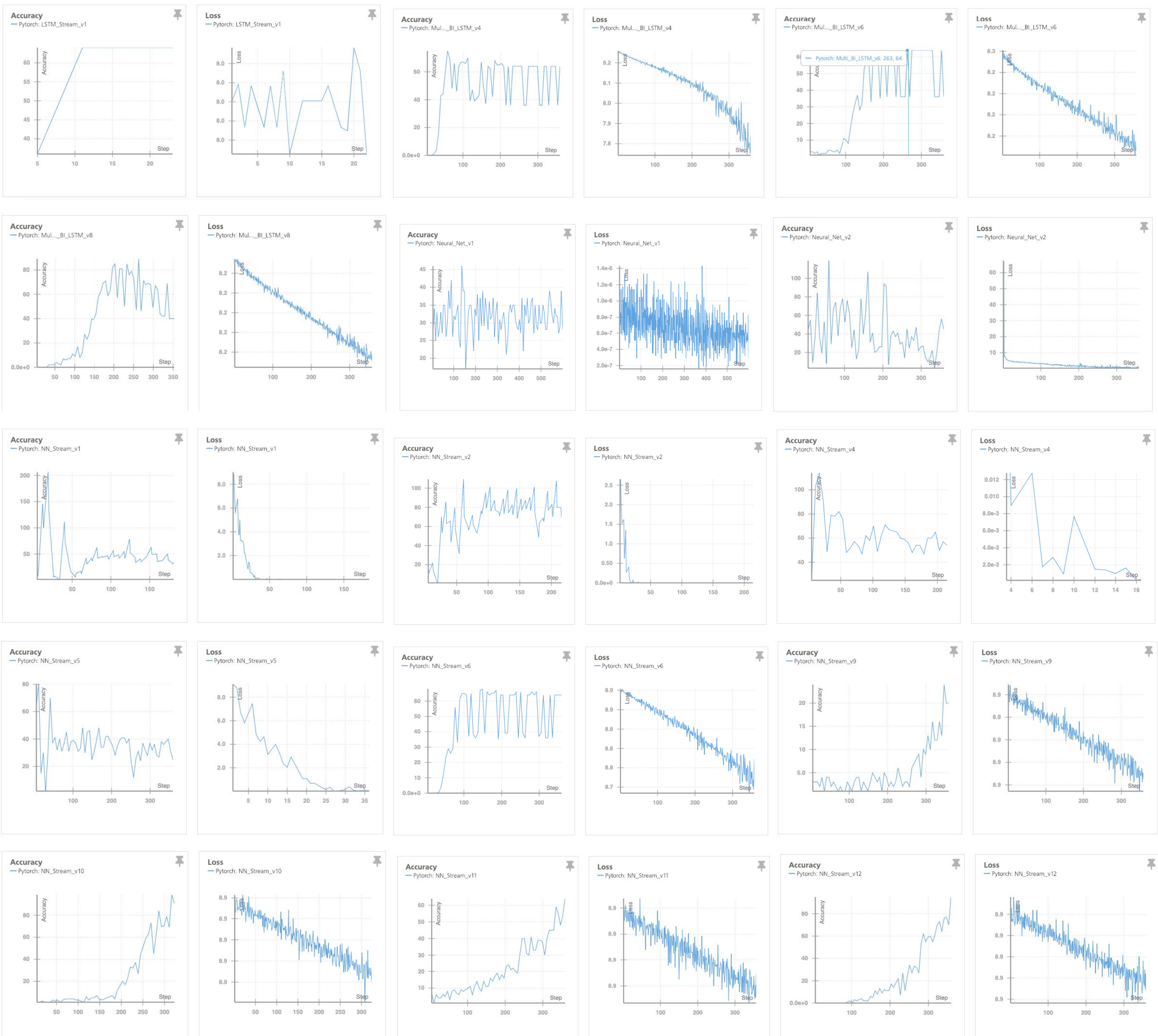


Figure 26: Above are lost and accuracy experiments done in wandb

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.

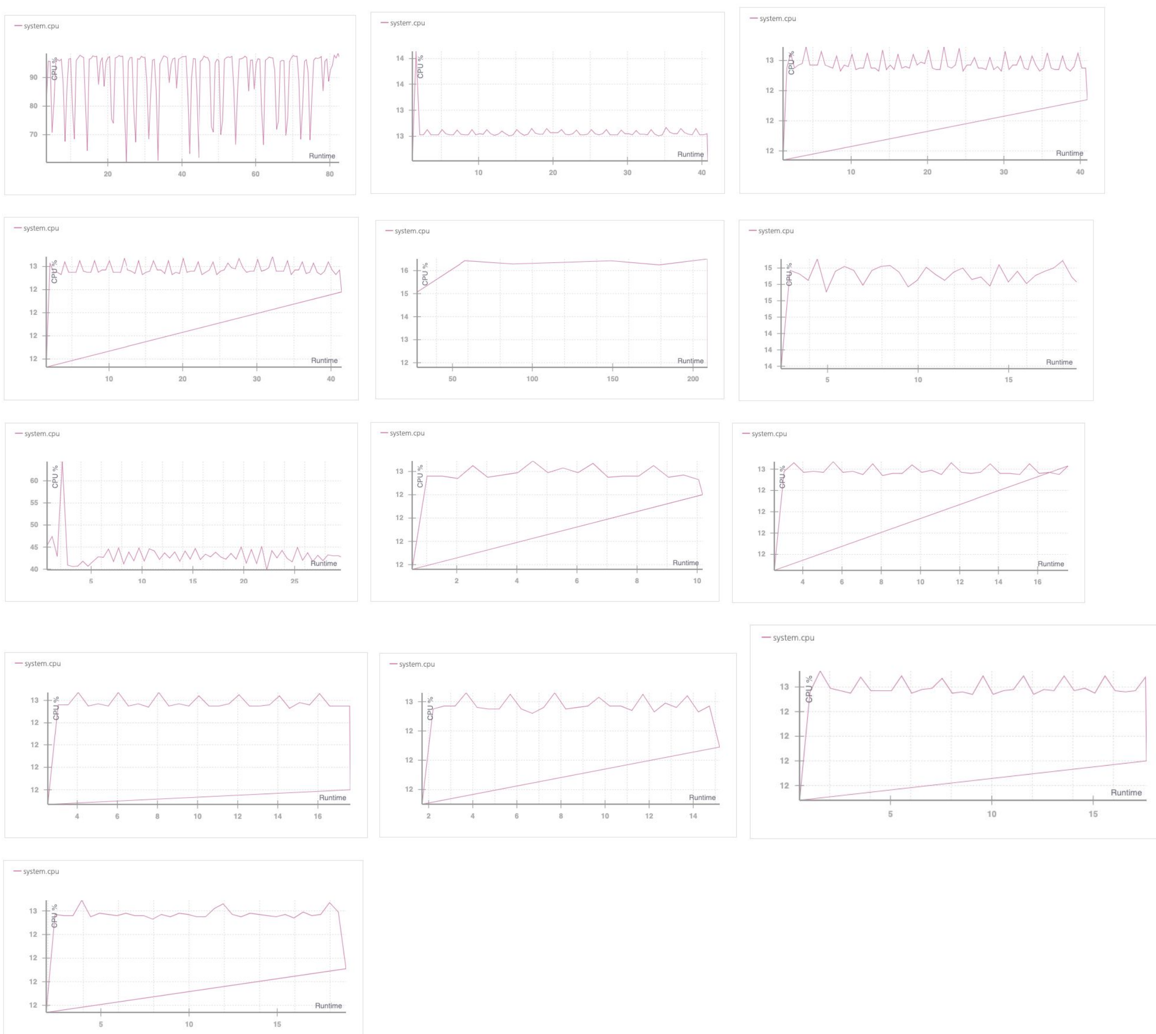


Figure 27: Above are CPU experiments done in wandb

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.

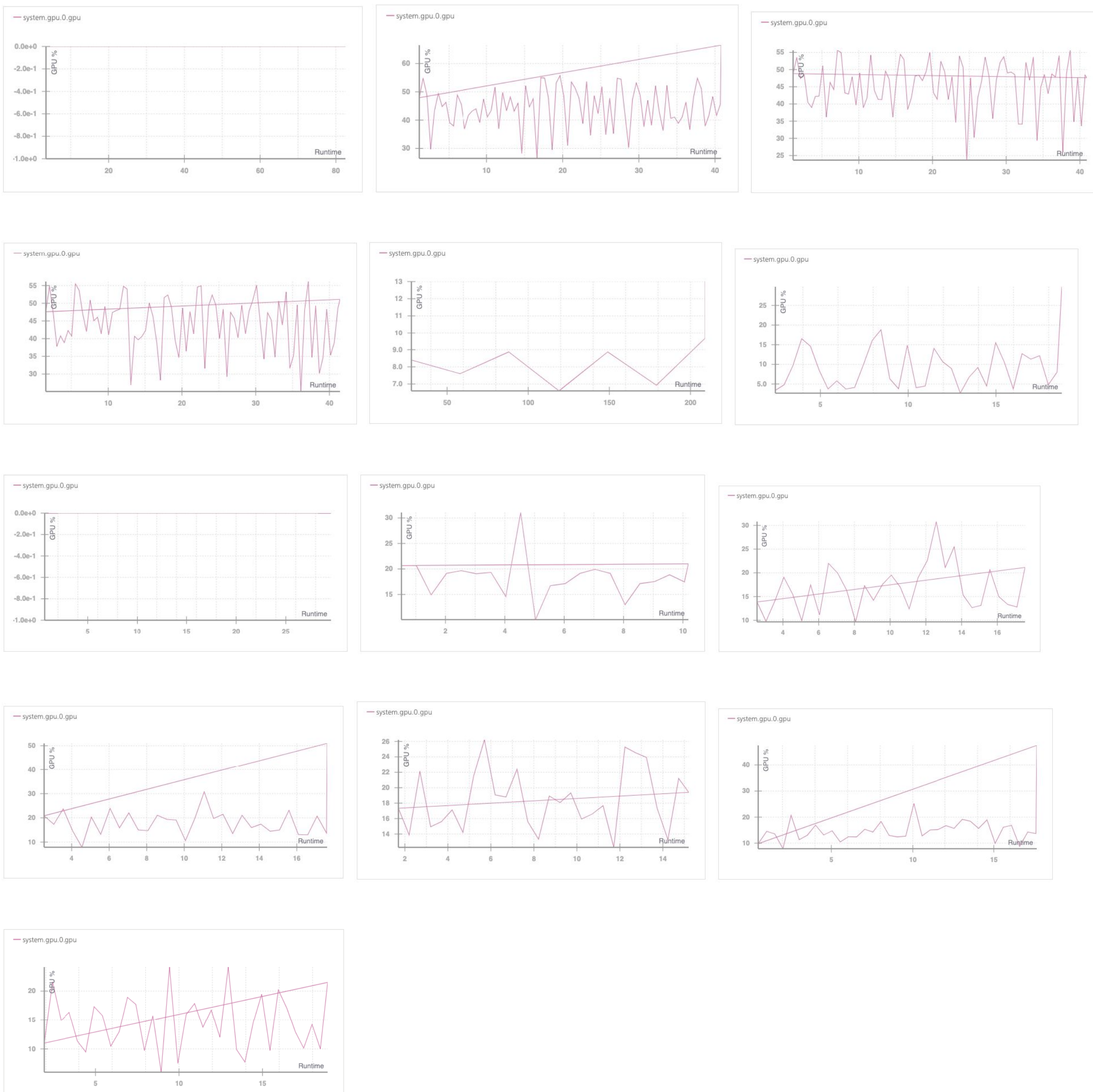


Figure 28: Above are GPU experiments done in wandb

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.

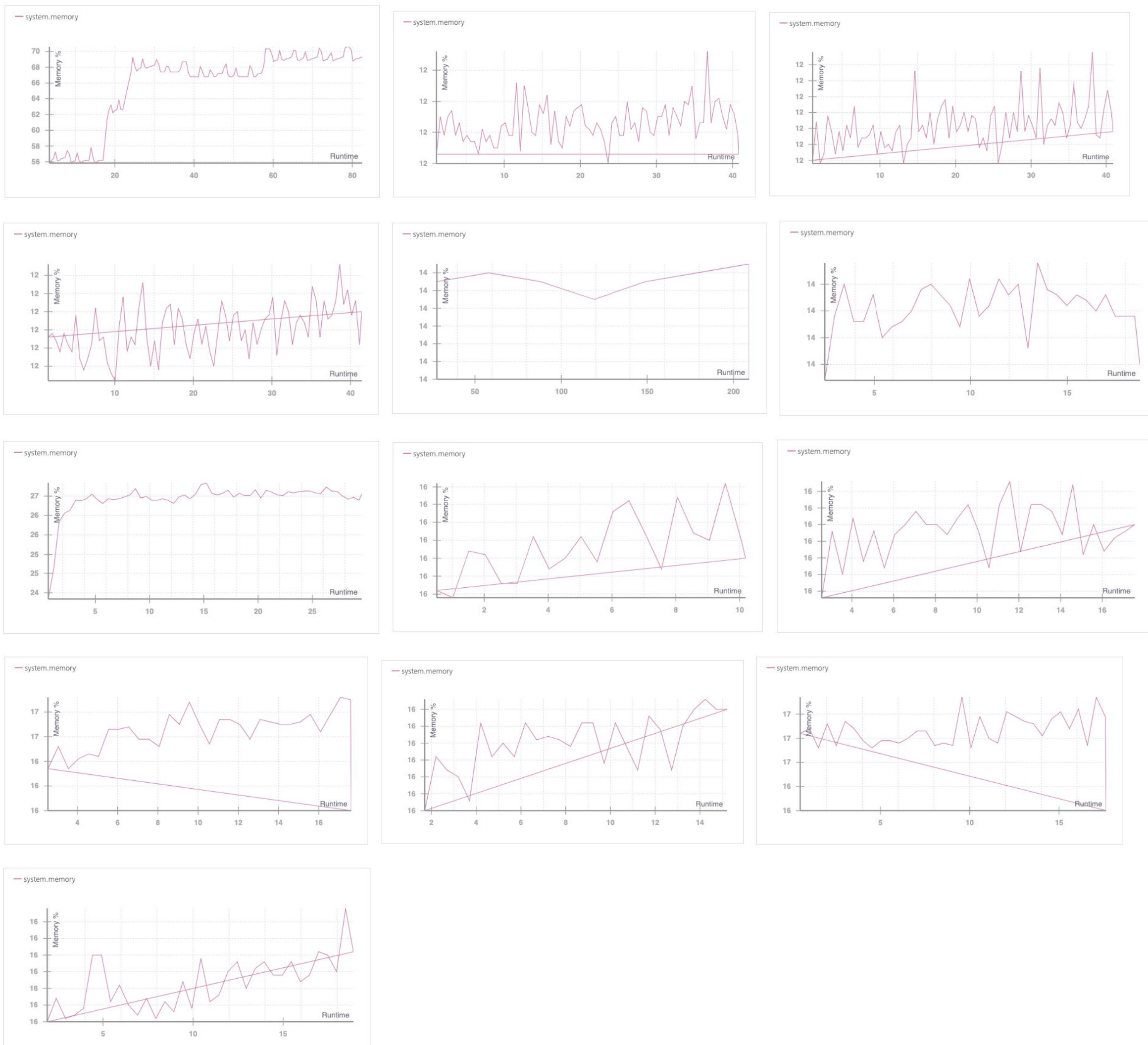


Figure 29: Above are hardware usage experiments done in wandb

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.

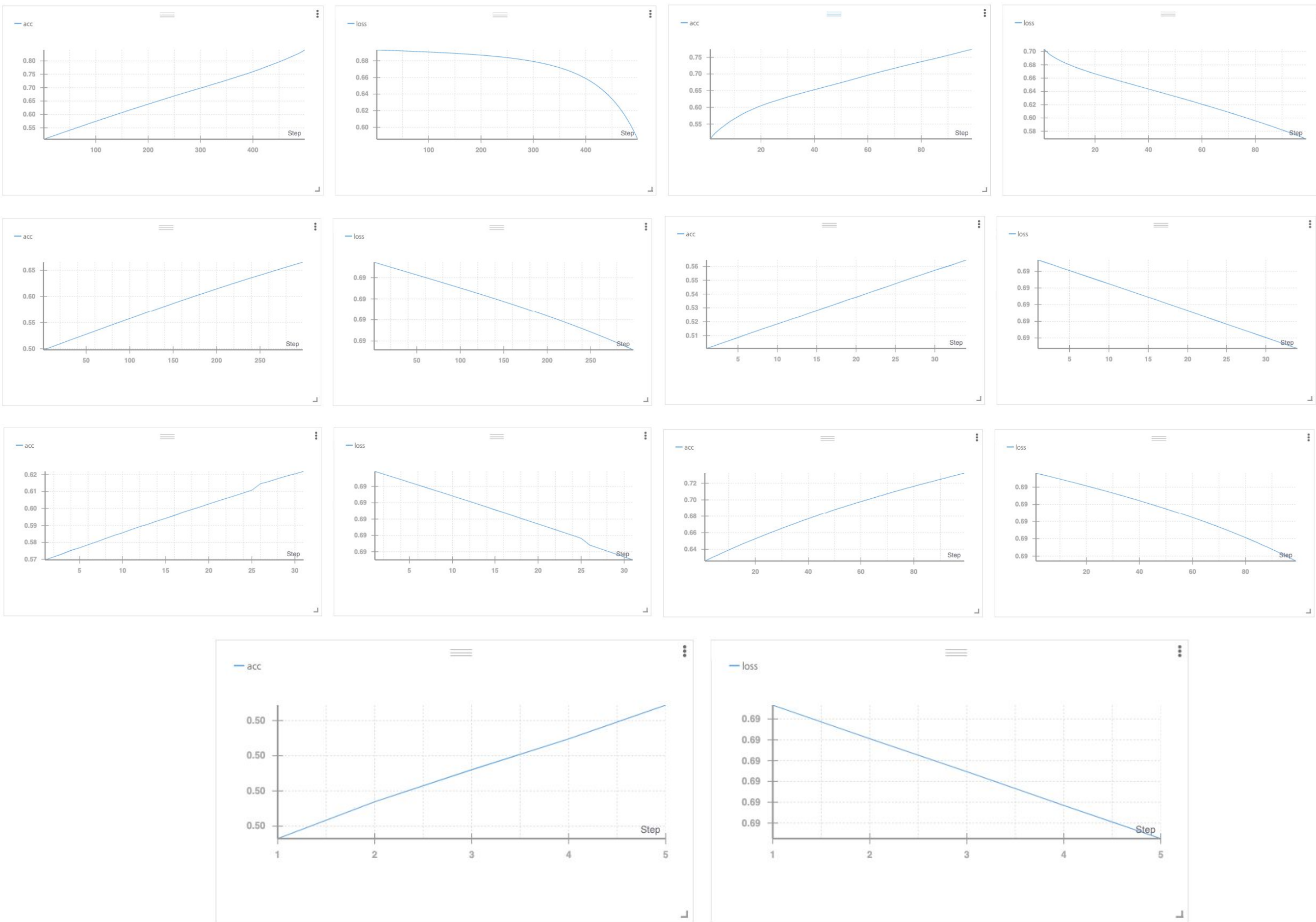


Figure 30: Above are my lost & accuracy (Keras) experiments done in wandb.

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.

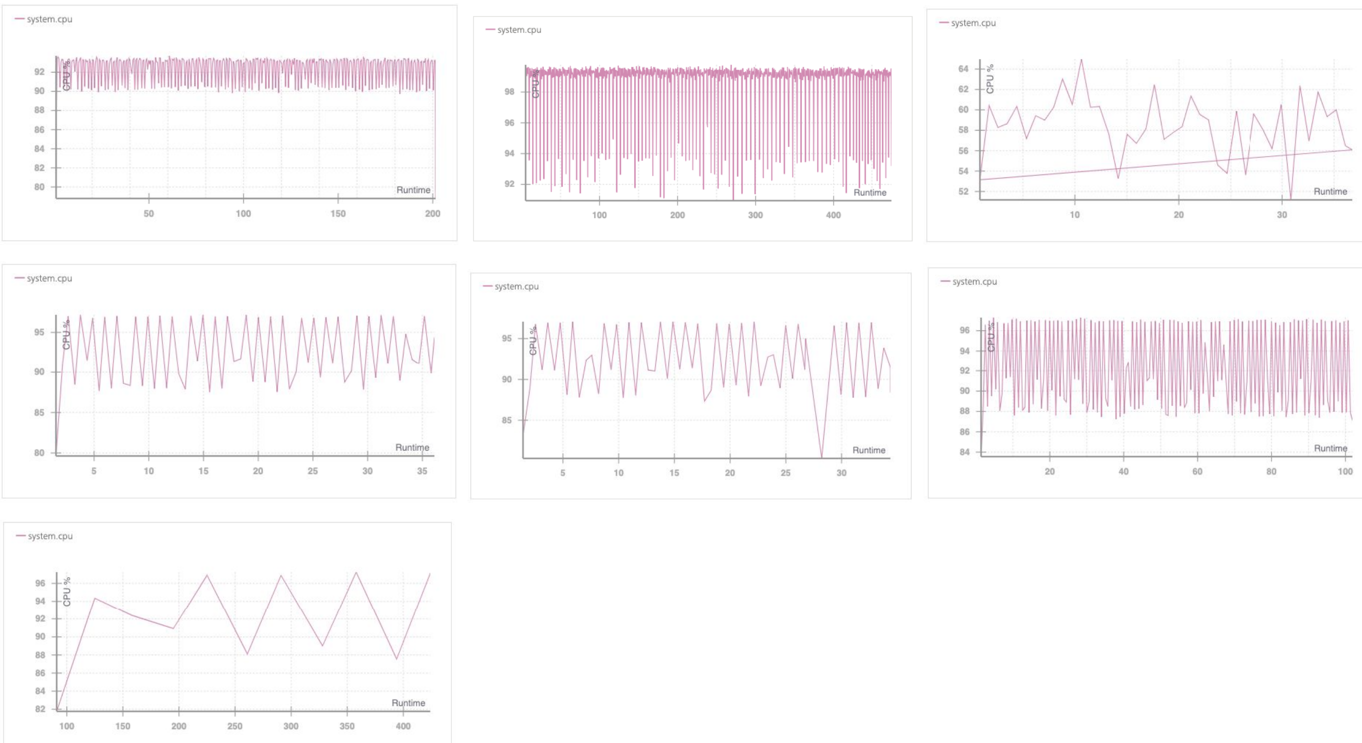


Figure 31: Above are system usage (Keras) experiments done in wandb.

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.

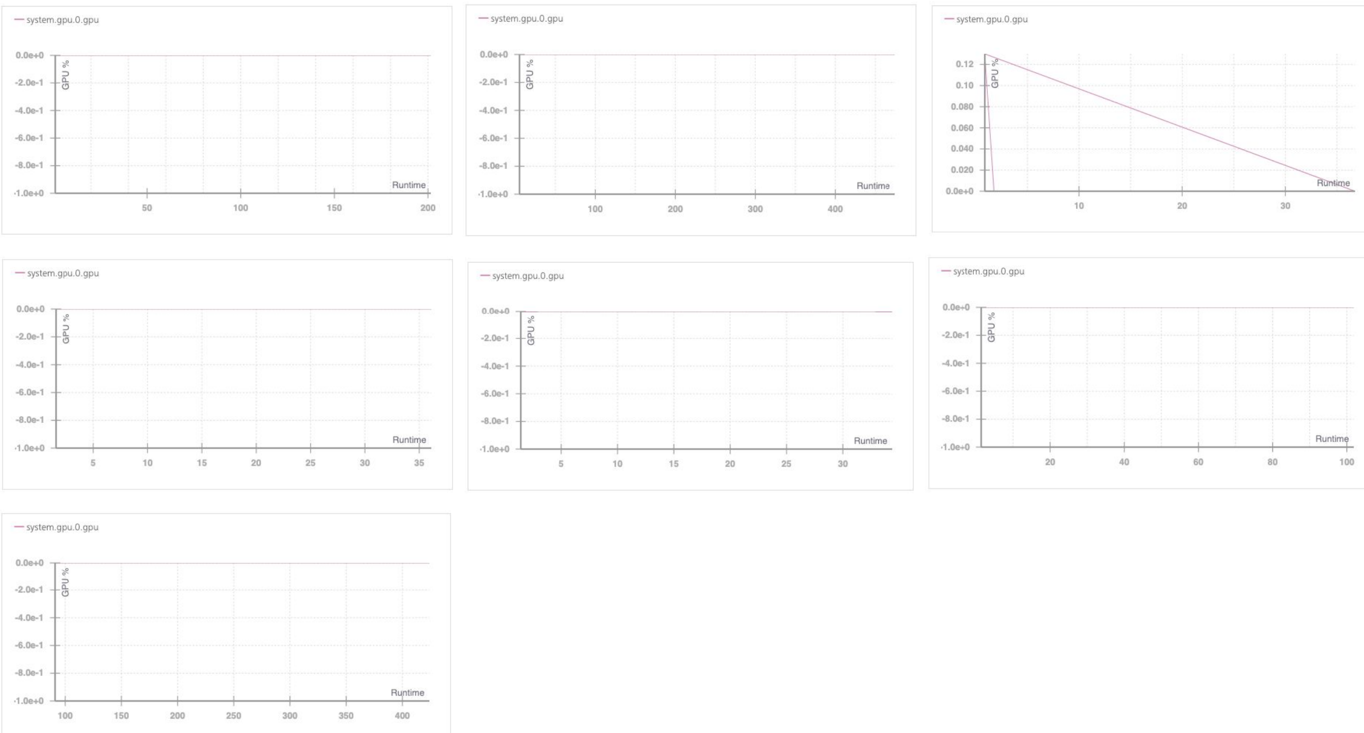


Figure 32: Above are GPU (Keras) experiments done in wandb.

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.

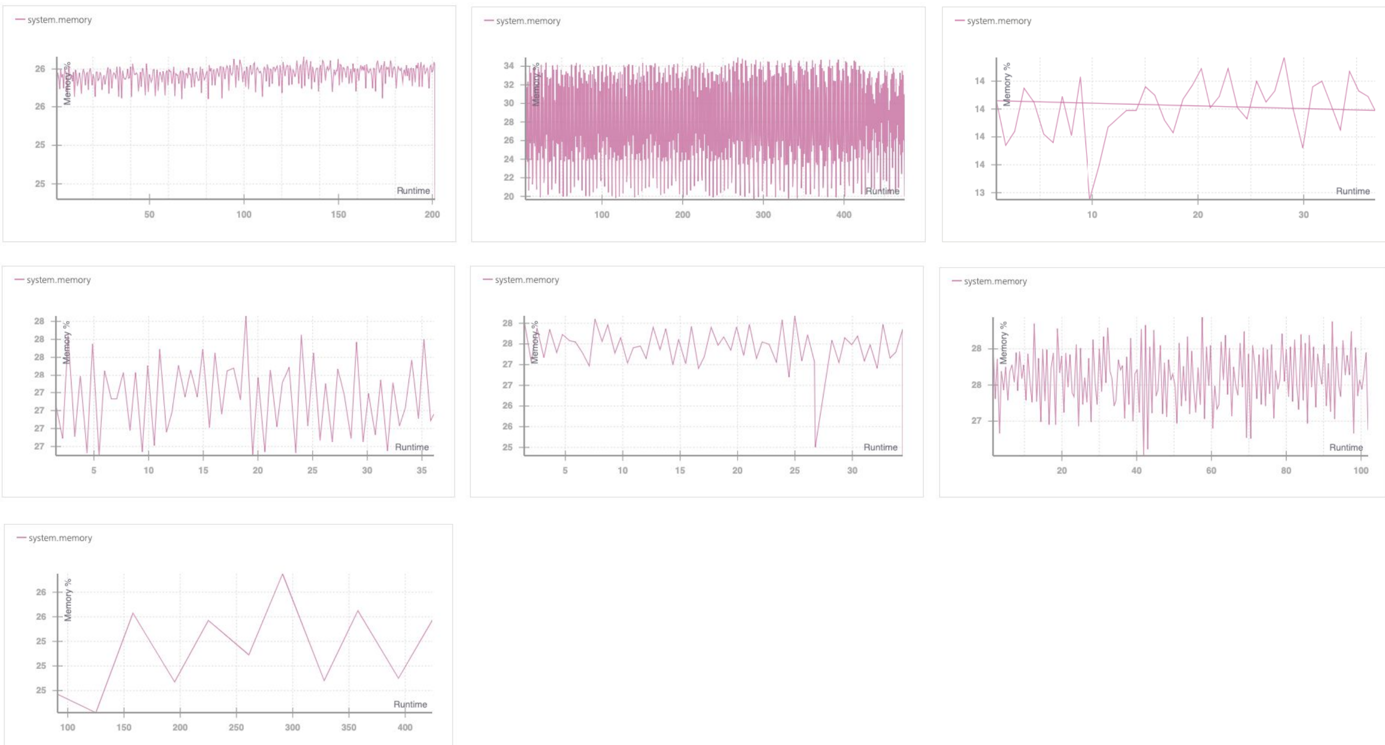


Figure 33: Above are system memory (Keras) experiments done in wandb.

Deep Neural Net

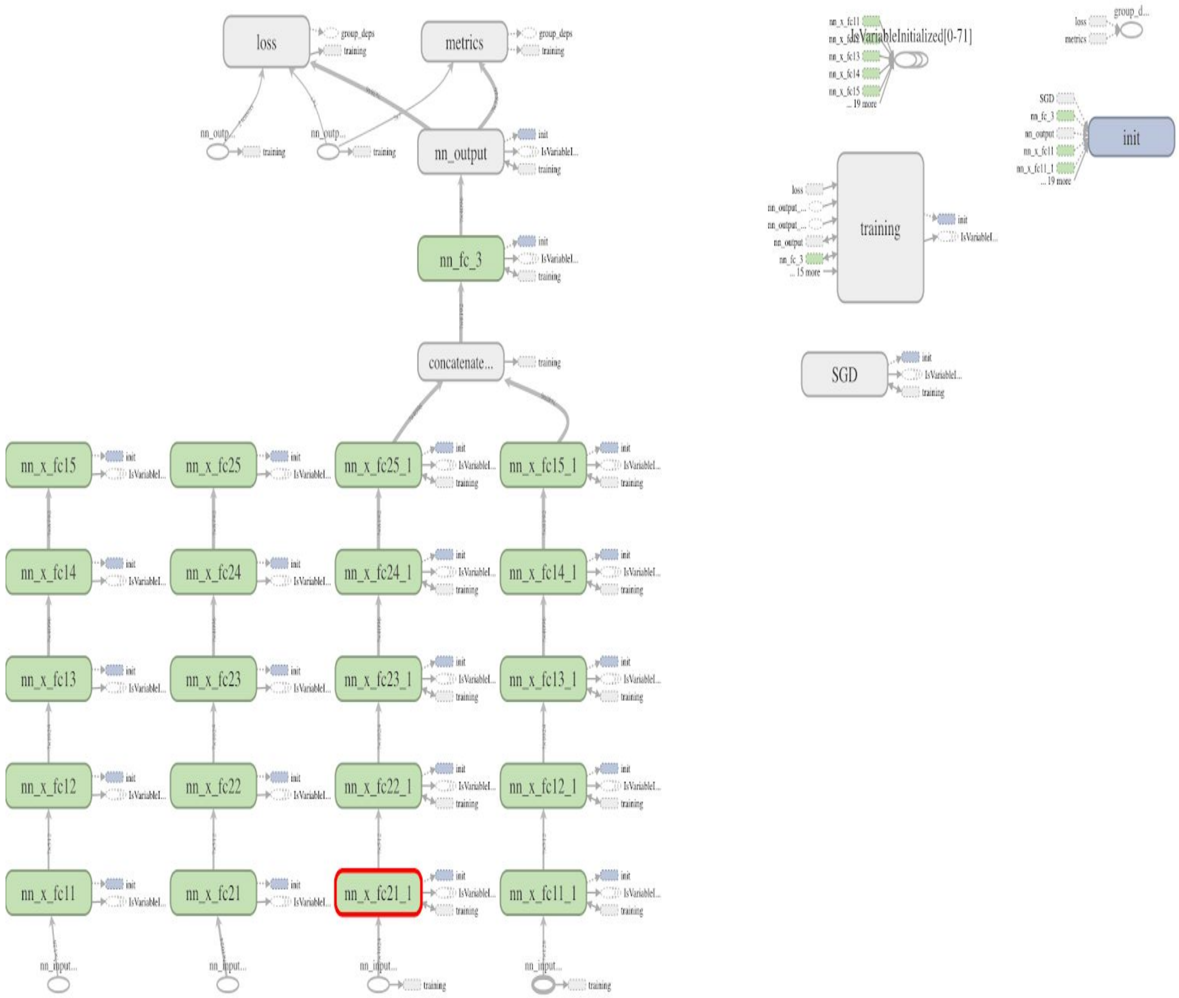


Figure 34: Above are system memory (Keras) experiments done in wandb.

Multi-Bidirectional LSTM

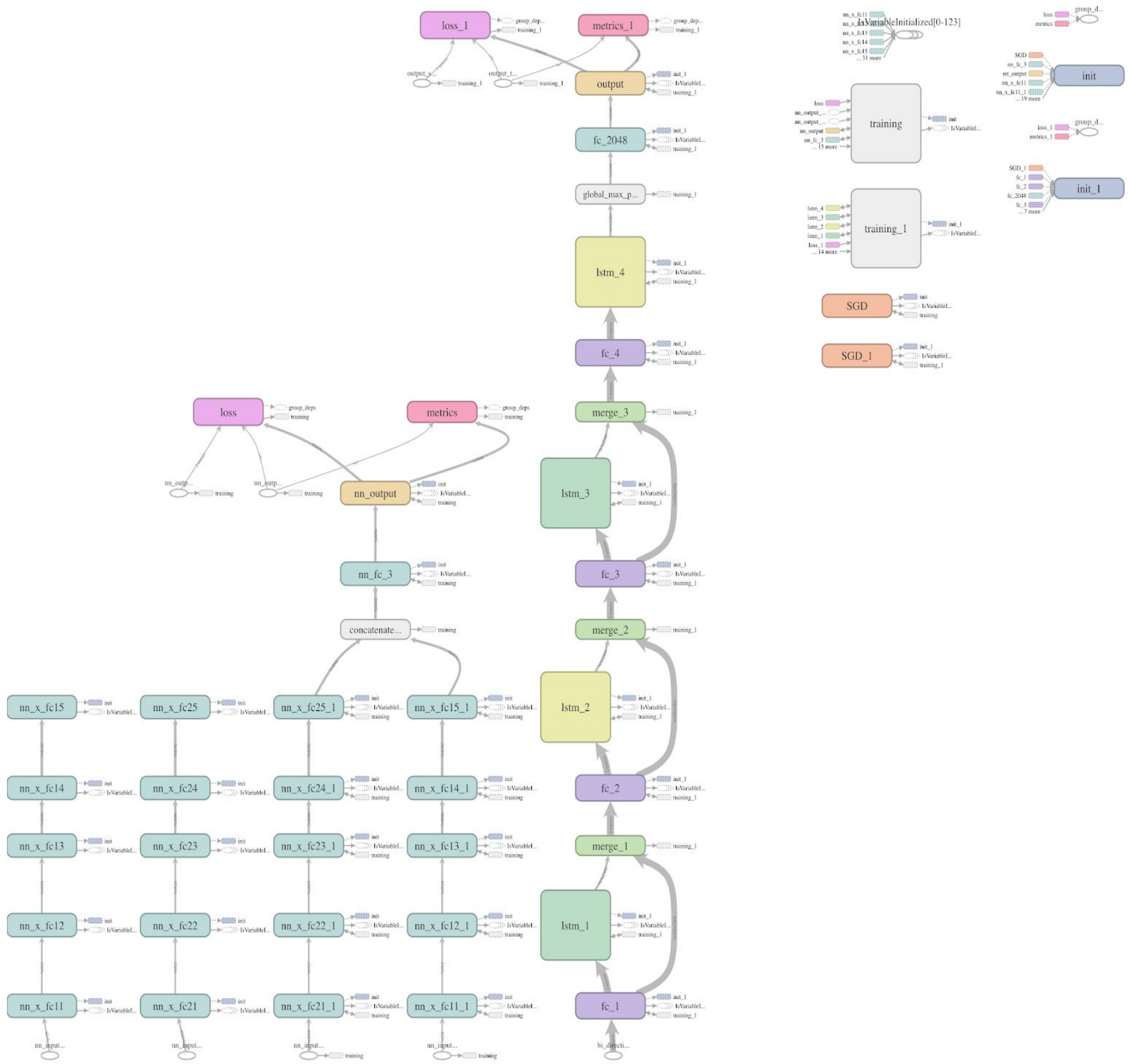


Figure 35: Above is a Bi-LSTM generated through tensorboard.

STREAM LSTM

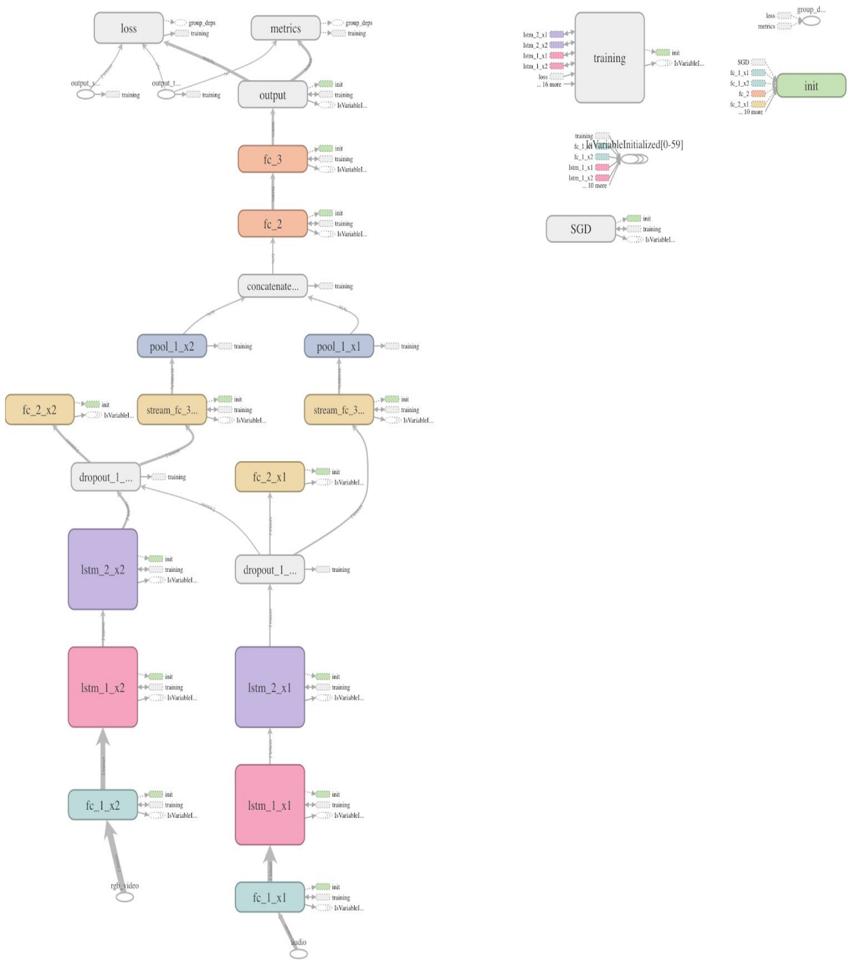


Figure 36: Above is a stream lstm generated through tensorboard

Neural Net LSTM Stream

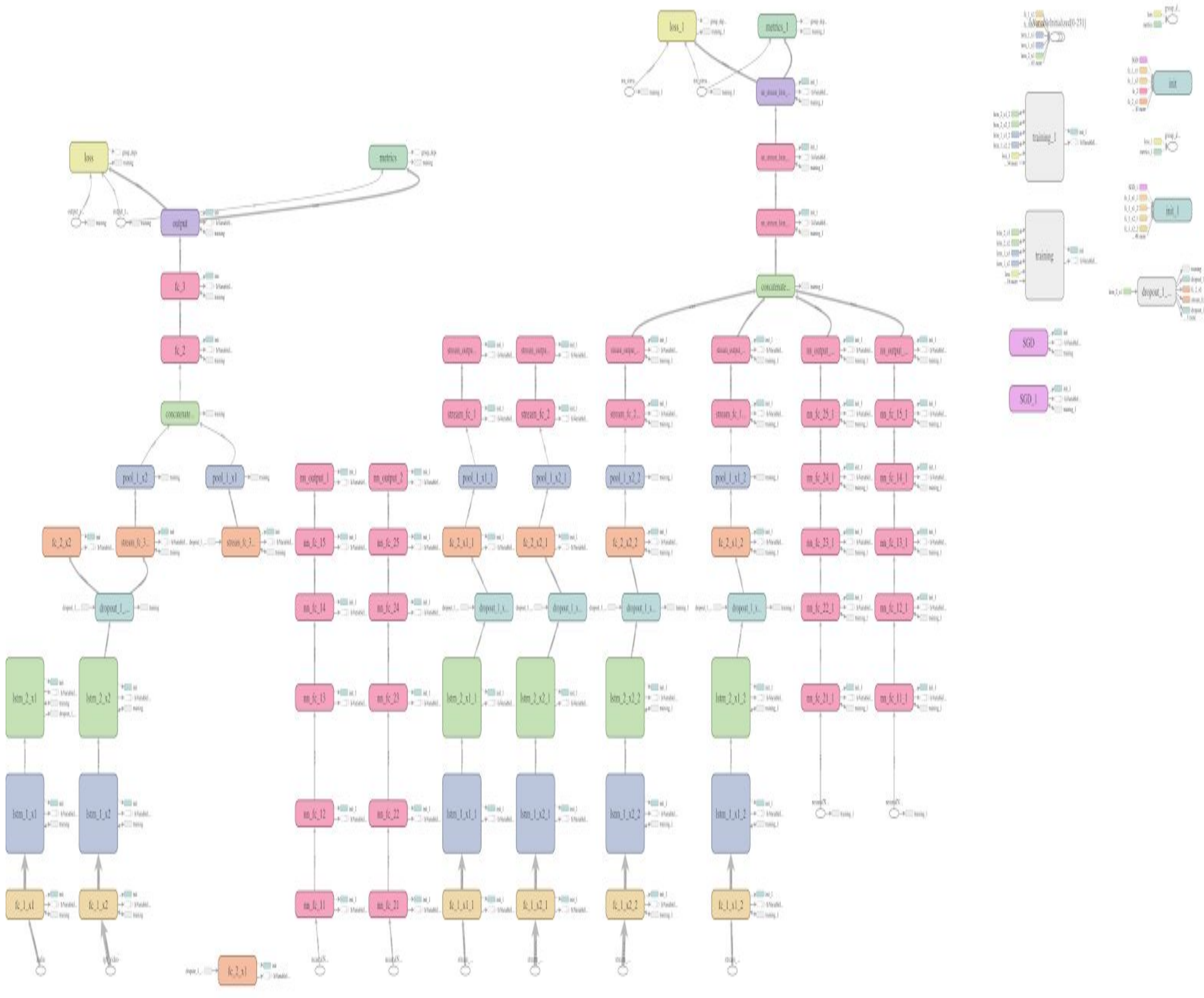


Figure 37: Above is a stream lstm generated through tensorboard