Drum Notator Documentation

Contents

1. Overview

This project was a university assignment where I was tasked to “self-learn” a topic and create something with it. I chose to learn TensorFlow and created this project. The following report was taken from my assignment to provide documentation for this project, and as a reference for myself for machine learning concepts found in relevant appendixes.

Please note that this GitHub repository does not contain audio files, or compiled network models, rather just the source code used for the application.

TensorFlow

TensorFlow is an open-source machine learning API created by Google. TensorFlow is very useful for analysing large datasets and is used in a lot of ways such as in Google Translate, Spotify’s song recommendation, and Airbnb’s image classification (TensorFlow, 2022).

TensorFlow adds ‘Tensors’ which are matrices (or multidimensional arrays) with various operations one can perform on them. Tensors are immutable objects that must contain a uniform datatype, and can either be constants, or variables (because tensors are immutable, variable tensors attempt to reuse memory from the original tensor when operations are performed). Using this, most TensorFlow machine learning models consist of a series of ‘layers’ that perform different operations on input tensors (representing a matrix of nodes) to produce output tensors. TensorFlow implements automatic differentiation that allows loss gradients to be found for each input-output in a model. Algorithms like backpropagation use this when fitting networks by passing in inputs and expected outputs, aiming to minimise loss by modifying layer parameters (or weights).

TensorFlow also includes ‘graphs’ which are data structures containing sets of operations and tensors to run TensorFlow code outside of python to allow portability, and faster speeds by applying optimisation algorithms during compilation.

2. Project Proposal

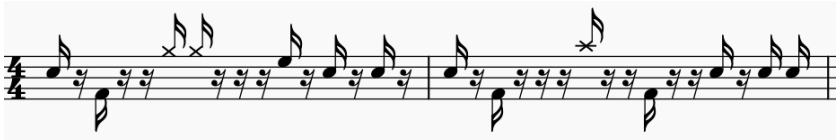
For my project I will use TensorFlow to create an application that attempts to musically notate what I play on the drum kit. The program should take in an audio recording of me playing the drum kit, and notate the rhythm I play, whist also identifying what drum I play each note on.

Minimum requirements:

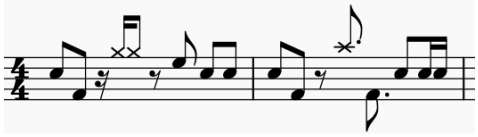
1. Obtain dataset, either via internet, or my own creation to train the network to identify which drum (snare, bass, toms, hi-hat, crash, ride) is being played in a sound clip.
2. Develop an algorithm (or train a network) to split a sound file up into different hits, so that each note is treated separately, and the individual drum can be identified by the network.
3. Train the network and produce a model that can then take in, and identify which drums are played in order from a user-created audio file. (As an extension if I can get it working, this may also include identifying multiple drums played at the same time).

Ideal requirements: *(attempted in order after minimum application completed. These requirements are lower priority as they have less emphasis on using TensorFlow, rather making my overall application better)*

1. Application identifies the rhythm that is played in a sound file, and creates a list of semiquavers, either played as a rest, or the identified drum. I can achieve this by looking at the time between each hit, given a BPM (Beats Per Minute).
2. Output converts this list to musical notation (in 4:4 time) either via an external application/API, or my own creation. Should look something like this:



3. Notation output identifies quavers, crochets, minims, semibreves, and dotted notes, and follows notation conventions of joining lines for quavers and semiquavers within the same beat. I.e., The above rhythm should look something like this:



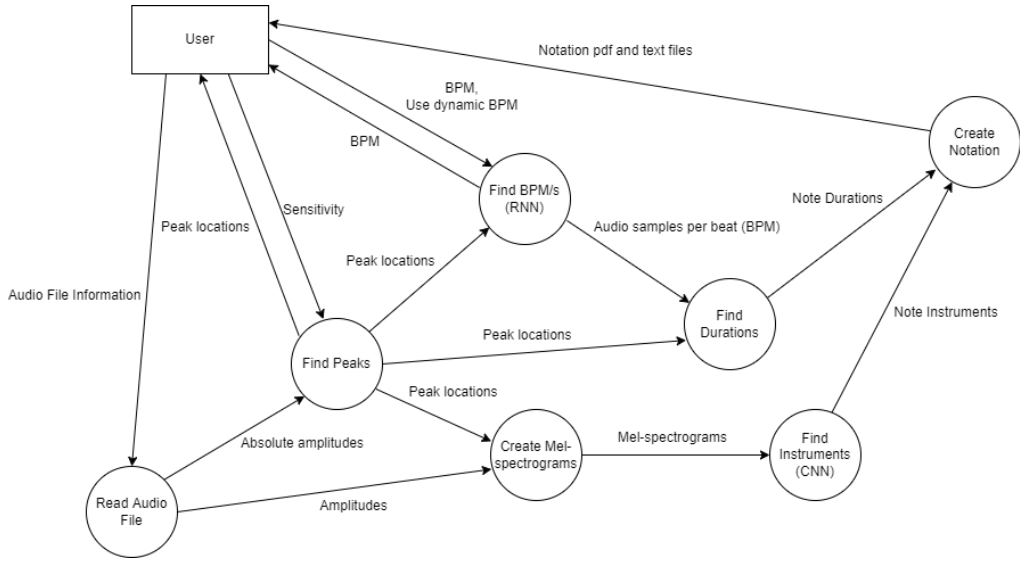
Both notation images were produced by myself using MuseScore.

3. Project Submission

Drum Notator

I have developed an application that notates what is played on the drum kit from an audio file input. With reference to my proposal, I accomplished the 6 requirements I created for myself in order, and then extended my application further.

Data Flow Diagram

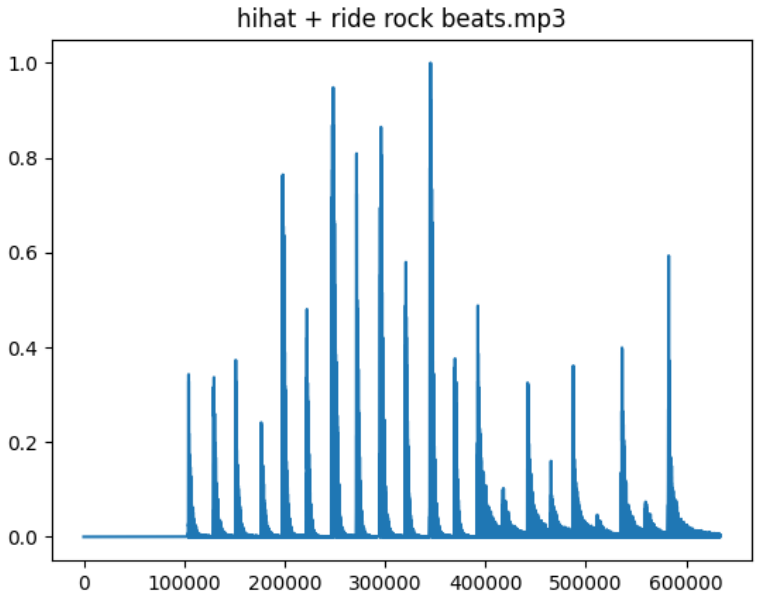


Data Input and Analysis

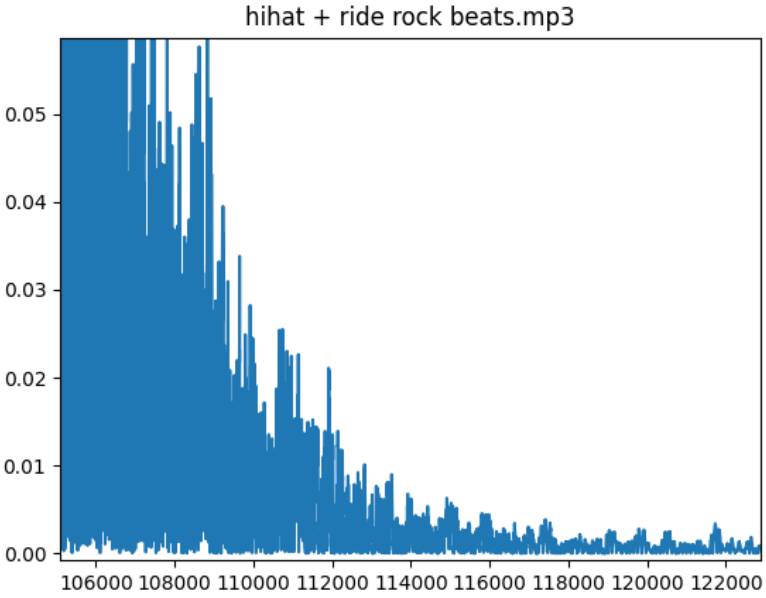
Reading Audio and Locating Notes (Peaks)

Inputting files is done using a separate 'TensorFlow I/O' module which allows a tensor of audio files to be returned. It is vital, firstly that the audio rate of files is always constant across both training and input files, as otherwise the Mel-spectrograms later inputted into the CNN (Convolutional Neural Network, see CNN Appendix) would be of differing size. The tensor which contains a list of amplitudes is then cleaned by firstly ensuring it is in float32 format (which I found could differ from the file entered). Next, its amplitude is normalized from a scale of -1 to 1, which helps account for volume differences so that drums played loud/softly, or differences in microphone volume, position of microphone, etc. are treated the same.

The input tensor is then used to identify peaks in absolute amplitude that are where each note is played in the file. Below is a Matplotlib graph to illustrate what an audio tensor looks like at this stage:

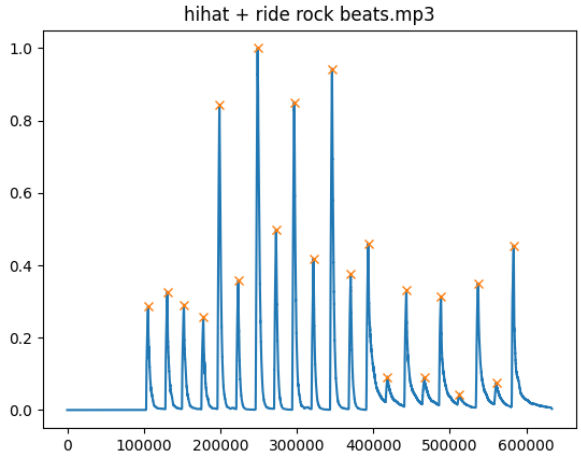


The SciPy library has a function that can identify these peaks. However, this function is not designed for noisy data. The following graph is zoomed in, (after the first visible peak, around the 100000th sample) to better illustrate the noise present:



Whilst there are other ways of accomplishing this, I decided to remove this noise by applying my CNN knowledge. This is achieved by using a 1-dimentional convolutional layer with a filter of exclusively ones that essentially smooths the amplitudes out; removing the

noise, while maintaining the large peaks that we are looking for. The data is then put into the SciPy function to identify the peaks and is plotted as below. This graph is presented to the user to ensure the algorithm is picking up desired peaks, but not unwanted peaks. This allows the option to change the sensitivity to help fine tune audio files. Here is an example of what is displayed to the user as identified peaks:



Identifying Instruments (CNN)

Once the peak positions are identified, Mel-spectrograms are generated at the start of each peak and for a certain (constant) duration afterwards. This is determined by the smallest possible note duration accepted, as overlapping images could lead to errors. Mel-spectrograms are frequency vs time images that represent audio and make use of the logarithmic Mel-scale to adjust frequency, making it a linear scale (Roberts, 2020).

These images are needed so that the audio data at each peak can be passed into a CNN. I decided to use this architecture over others like RNNs (Recurrent Neural Networks, which are commonly used for audio-based data; see RNN Appendix) because I anticipate the network to function by examining frequencies present, and other possible attributes like tone and resonance in the data. While the Mel-spectrogram visually contains these features, using an RNN to pass in amplitudes at each point would therefore (likely) require the network to do additional processing to determine frequencies present. Because of TensorFlow's easy-to-use spectrogram functions, and my immediate success using a CNN due to prior experience of it, I did not test the network using an RNN.

Unlike other CNNs, in which data input could be one of a set of possible categories, this data can have multiple classifications (i.e., if one was to play the bass and snare drum at the same time). To implement this requires a more complicated network, which is harder to train, which required me to record training data representing multiple instruments at once.

Identifying rhythm

Finding Durations

Once we have the positions of peaks in absolute amplitude, we can use the distance between these peaks to determine the duration of a note as a fraction of a beat, given the SPB.

We will assume the first four notes in the audio are a count in, (as crochets) and will be used to find the SPB, being excluded from the final notation. This method is mostly accurate; however, it assumes the drummer is 100% on tempo, which is not usually the case. Therefore, a small work-around was made to convert this to BPM and check that it is correct with the user.

Rounding Durations

As mentioned before, there will always be slight 'errors' in any drummer's timing, which can be errors, but also can be associated with 'feel' and how a piece is played. Therefore, the note durations must be rounded to their closest associated note to attempt to understand what is being played. Errors associated with this rounding get more frequent the more subdivisions in a beat are allowed (i.e. the fastest note allowed).

Subdivision works by dividing one beat into multiple smaller notes, metrically a 'crochet' is worth 1 beat, 'quaver' is 1/2, and 'semiquaver' is 1/4. Therefore, using semiquavers as the smallest division means the smallest error in the drummer's duration is 1/8 (half the smallest subdivision).

The rounding error remainder is used in the next note to prevent accumulated rounding errors. These errors might occur when the drummer goes out of time briefly (by perhaps rushing during a fill/solo), and then returning to the correct tempo. By feeding these remainders forward essentially means that the total accumulated duration of the score, rounded to the nearest subdivision, is equal to the sum of each rounded durations. This is important because the drummer will return to the correct tempo which is aligned with the total duration from the first beat; without considering this, rounding errors can accumulate and can eventually offset the duration by small amounts (like semi-quavers), which makes everything preceding the point of offset incorrect.

Triplets

Subdivision of normal notes is base 2 with reference to the above explanation, however triplets require a base of 3, and therefore crochet-triplets are worth 2/3 of a beat, and quaver-triplets are 1/3.

An issue that triplets cause is that the smallest error using quaver-triplets and semiquavers as the smallest notes drops from 1/8, to 1/24 (as half of the difference between 1/3 and 1/4). However, assuming quaver-triplets are the smallest base 3 subdivision, a beat cannot contain both base 2 and base 3 divisions (which will be referred to as quaver and triplet divisions). Therefore, at the start of each beat that will be subdivided into multiple notes, the sum of errors in quaver and triplet representation is calculated to determine which is more accurate.

The Tempo Problem (RNN)

Finding the durations of notes can only be done if tempo (measured in BPM) is known. However, drummers are not always playing to metronomes, and some songs are recorded without metronomes and thus have dynamic BPMs throughout the song. I made three attempts to solve this problem by calculating a moving BPM to determine note durations, and unfortunately none of them were successful (see test 4 in screenshots section). The attempt currently in the code makes use of a dynamic RNN using LSTM layers (see RNN Appendix) that attempts to estimate a moving BPM at each peak, and regard to distances between it and the next peak. The way I trained the network was by playing some rhythms and then separately recording myself playing crochets along to those rhythms to measure BPM.

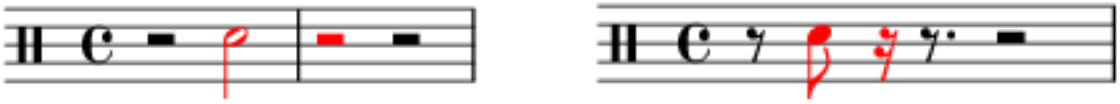
At each peak, the RNN takes in the distance (in samples) to the next peak, and the last predicted SPB (samples per beat, which is proportional to the BPM, but more useful when considering audio samples), or the initial SPB if the peak is the first, in the audio, after the count in. It will determine and output the SPB of the current peak. The SPB is necessary because it is arbitrary if 4 equidistance peaks are crochets at say 60BPM, quavers at 30BPM, minims at 120BPM, etc.

To effectively pass in the stated inputs, all inputs and expected outputs will be divided by the initial SPB. The effect of this is that the normalized initial SPB will always be 1, and preceding timestamps will pass forward previous information, which ultimately means only a single input feature is required. Outputs are therefore also expected to have been normalized, meaning the SPB outputs of the network must then be multiplied by initial SPB once again.

Creating Notation

Adding rest notes

Durations must follow conventions when considered in the context of beats and bars; a note cannot extend beyond the bar it is within, and a subdivision cannot extend beyond the beat it is within. For example, a note duration of 4 beats positioned on beat 3 of bar must convert to a minim, followed by a minim rest in the next bar; and a note duration of 3 semiquavers positioned on beat 1+ (the third semiquaver of beat 1) must convert to a quaver, followed by a semiquaver rest on the next beat:



Notes can have duration of more than a beat, these include minims (2 beats) and semibreves (4 beats). The program assumes there are always 4 beats to a bar.

LilyPond Notation

Notating the final product is done using 'Mingus' library which provides a way to notate music in Python. However, it is rather restrictive in what it allows, and is not designed for notating a drumkit. The library however only provides an interface to use 'LilyPond', which does provide drumkit notation functionality, and Mingus fortunately contains a function to export a string directly to LilyPond to convert it to a pdf.

Stem Directions

LilyPond allows multiple 'voices' to be created that allows up and down stem directions to be used. However, there are very loose conventions (usually dependent on personal preference) for drum notation as to when instruments are to be put up or down and is usually dependent on the broader context of the score:

1. During grooves (like rock beats, swing beats, etc.) it is best to put the bass, snare and floor tom down and everything else up.
2. During fills everything goes up, unless a bass drum is played at the same time as other drums in which case it should go down.
3. If at any point a hi-hat is played with the foot (which is usually the case if it is played with another cymbal, or floor tom), it should be notated on the same space as the bass drum (and its stem should go down).
4. There should be consistency in stem directions within sections of the score

Converting Durations to Lilypond's format

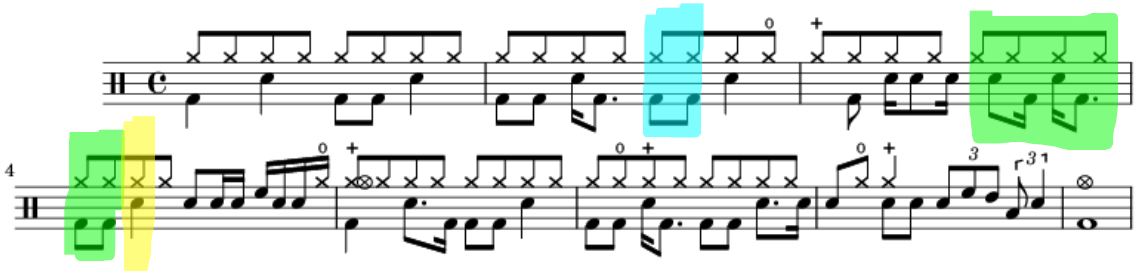
During other points in the code, it makes most sense to represent note durations as floating-point values for the amount of beat/s it takes up so that meaningful arithmetic can be done (see above sections). However LilyPond requires an integer that essentially represents 'how many of this note can you fit in a bar', (i.e. semibreve is 1, crochet is 4, quaver is 8, etc.). This value is the same for triplets with *crochet* triplets being 4 and *quaver* triplets as 8, however the triplet must be wrapped in a tag to separate it from other notes. Furthermore, LilyPond contains functionality for dotted notes (worth 1.5 times the note, i.e. A dotted quaver is worth 3 semiquavers) by adding a '.' After this integer.

Conversion between these two representations could be done with a dictionary, however for future development that may potentially involve other notes, it can be done generically using some mathematics involving logarithms to round notes to their nearest non-dotted duration. Using this the note's value is determined, then if there is a remainder, make the note dotted.

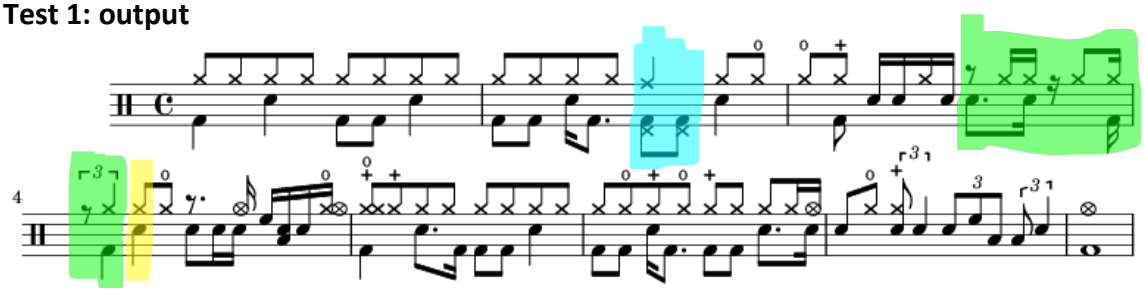
4. Screenshots

To test the success of the program I have recorded several takes of myself playing the drums, to see how accurately the program notates what I play. To illustrate where the program made mistakes, I have corrected the original outputs to the expected outputs which are compared below.

Test 1: expected output

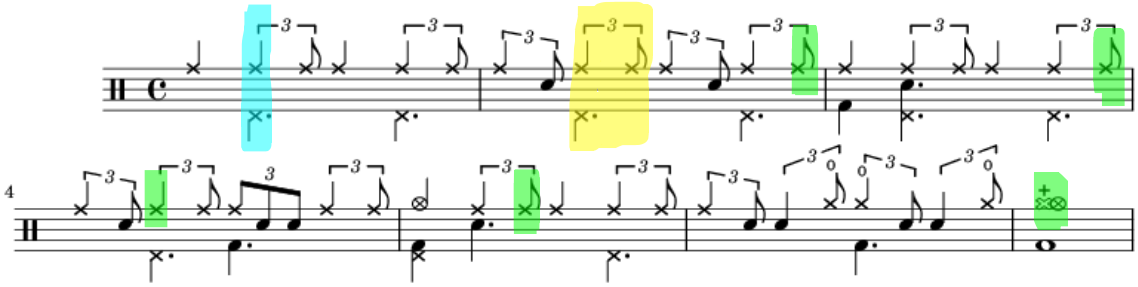


Test 1: output

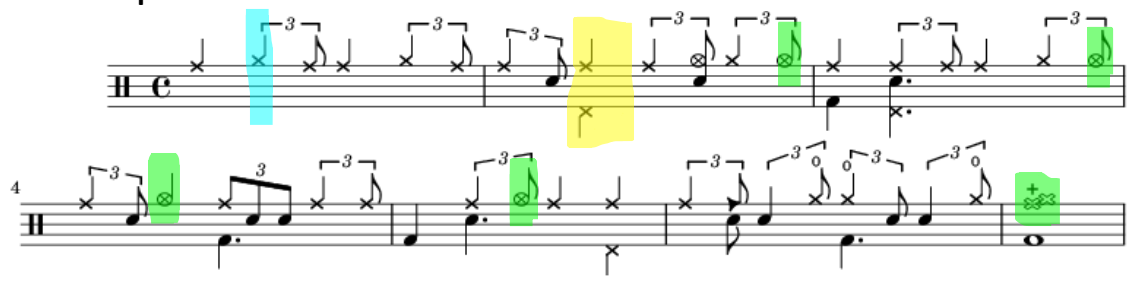


Test 1 was very successful, with only a few minor errors from the program. Some things to consider is that a small error can cascade into a larger percived error. For example, in beat 3 of bar 2 (highlighted blue) the program incorrectly identified a ride, however correctly identified the two hi-hats and bass drums. However, because a ride is present, the notation algorithm puts the two hi-hats below the bass drums which makes the error appear larger than it actually is. The program had most errors in bar 3 (highlighted green), where I was slightly out of time, but correctly returns to the correct notation in beat 2 of bar 4 (highlighted yellow) when I return to the correct tempo.

Test 2: expected output

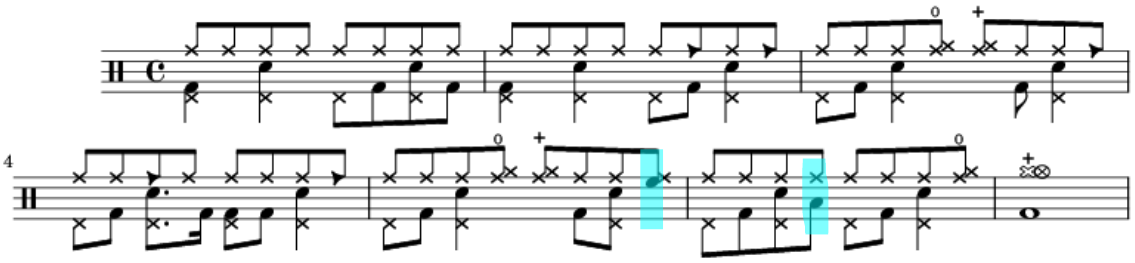


Test 2: output

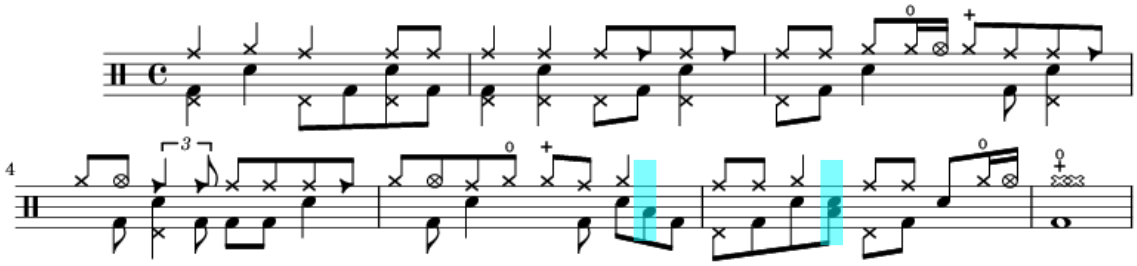


Test 2 was also rather successful. The main reason for errors is because the ride cymbal is extremely quiet. Almost all errors here are due to either the peak identification not picking up on the ride (like in beat 2 of bar 2, highlighted blue), or the CNN not identifying the ride, while correctly identifying all other instruments played (like beat 2 of bar 1, highlighted yellow). Another observation is that it often mixes up ride and crashes (highlighted green), which was expected as these are very similar instruments.

Test 3: expected output

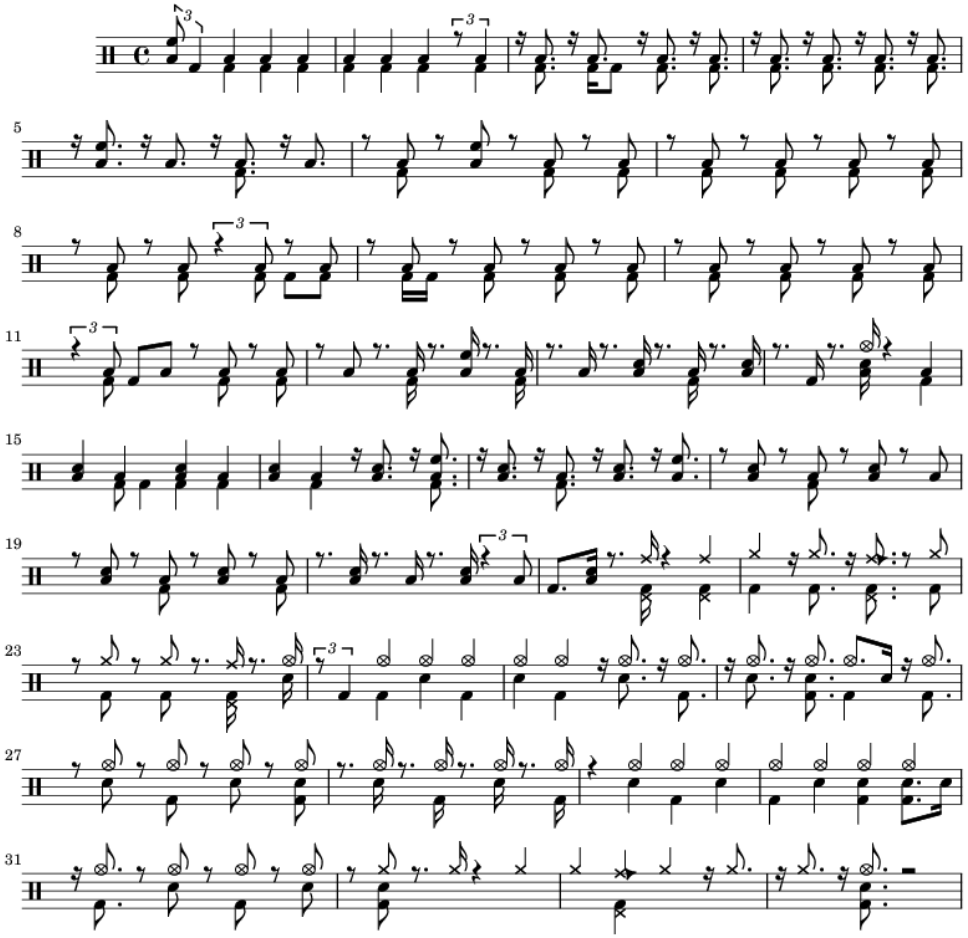


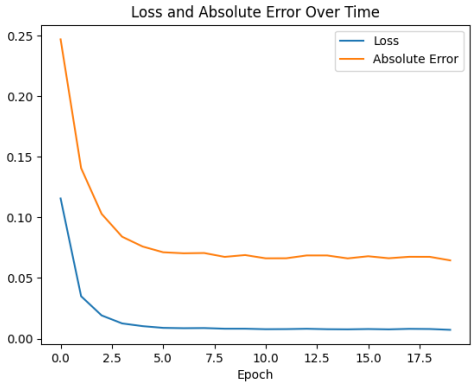
Test 3: output



Test 3 was designed to examine the limits in terms of instrument identification. As with test 1, it was very difficult for the program to identify rides which constitutes most of the errors. As can be seen, the program had trouble identifying toms (highlighted blue), which like cymbals can be easily confused (I had trouble myself working out which toms I played while producing the corrected notations).

Test 4: output

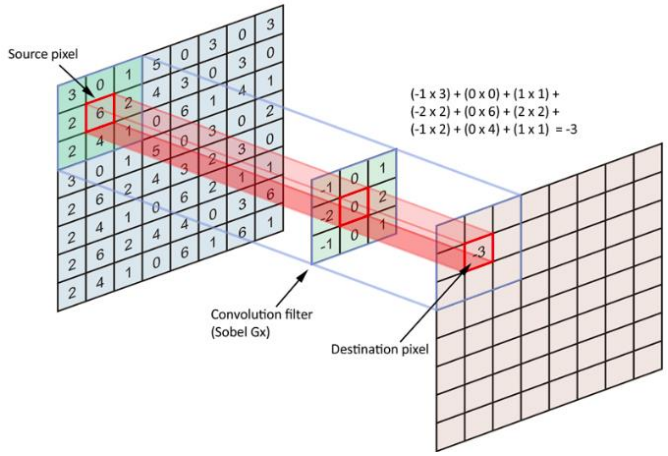




Test 4 was played to the song 'Seven Nation Army', which does not have a constant BPM (as it was not recorded to a click track). This test was supposed to evaluate the dynamic BPM functionality with the RNN. Unfortunately, in training, the RNN experienced the vanishing gradient problem (see RNN Appendix) which has been graphed to the right. I found after extensive trial and error, that the lowest absolute error I could achieve was around 0.06 (meaning 6% of the BPM). Unfortunately, this is too high for the model to accurately produce notation (as is demonstrated above), and I did not attempt to notate the expected output for this test as almost all durations are supposed to be crochets, making it almost entirely incorrect. Because using an RNN was my third attempt at creating a way of solving the dynamic BPM problem, which was not in my originally specified requirements I decided to focus time on other aspects of the project.

5. CNN Appendix

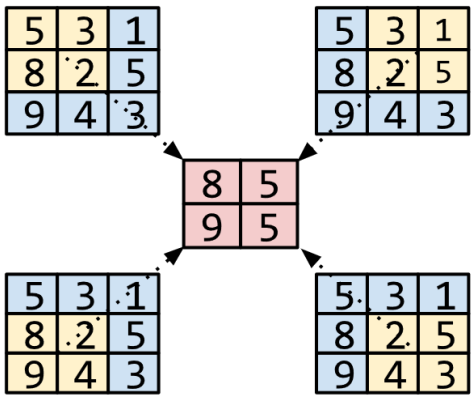
Convolutional layers:



Convolutional layer, using 3x3 filter (Cornelisse, 2018)

Convolutional layers work by multiplying a filter over the input matrix and summing the results, to form each element in the output matrix (Google Developers, 2021).

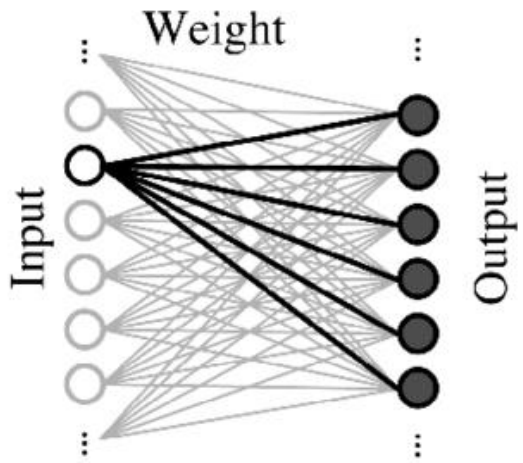
Pooling layers:



Pooling layer, using maximum of slices (Google Developers, 2021)

Pooling layers are a way of summarizing a matrix (tensor), by splitting it into smaller slices and compressing each of those into a single scalar, which forms an element in its output (Google Developers, 2021). These layers are often used after convolutional layers.

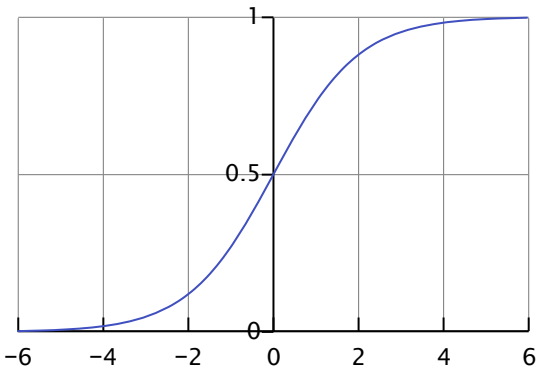
Dense layers:



Dense layer illustration (Ando, Takamaeda-Yamazaki, Ikebe, Asai, & Motomura, 2016)

Dense layers map each input node to each output through a function (Google Developers, 2021). Within TensorFlow this is achieved with only one function representing all nodes using matrix multiplication and addition.

Sigmoid activation:



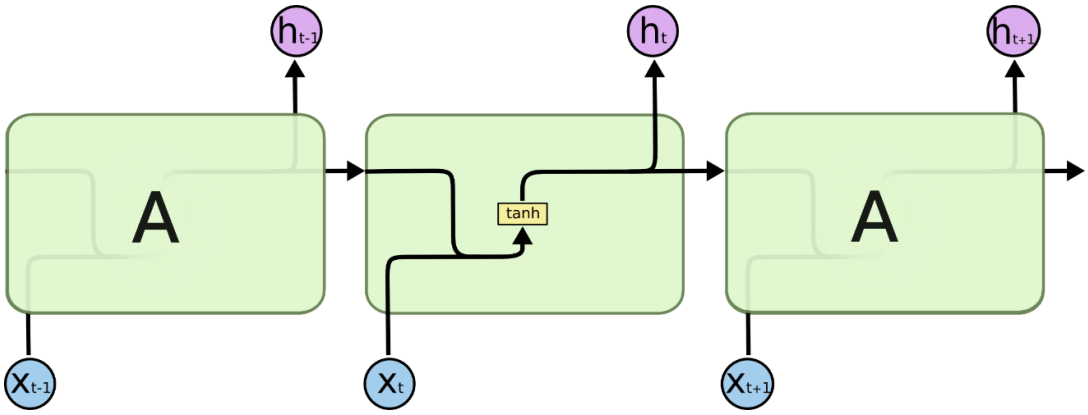
Example of a sigmoid activation function (Richards, 2008)

Sigmoid activation functions which are applied on the output of a layer to convert it to a value between 0 and 1 (Google Developers, 2021). Other activations can be used to achieve different output forms; this activation produces a number between 0 and 1, which could be used to represent a probability.

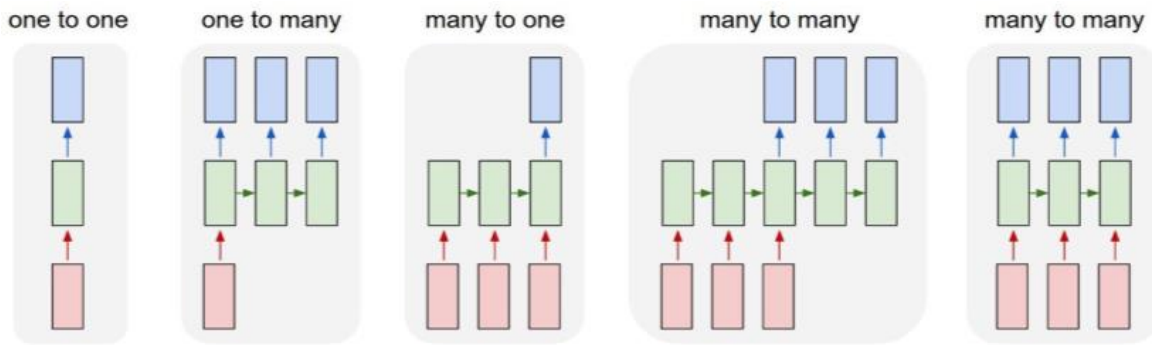
6. RNN Appendix

Recurrent Networks

While feed-forward networks (like CNNs) enforce a one-to-one input/output sequence, Recurrent Neural Networks (RNNs) behave differently by allowing the network to run multiple times, while retaining information from previous iterations through states. This process (using a tan activation after the hidden layer) is shown below:



RNN structure. Image cropped from (Jha, Long Short Term Memory [Image], 2016)



Implementations of RNNS. Image cropped from (Jha, Recurrent Neural Networks [Image], 2016)

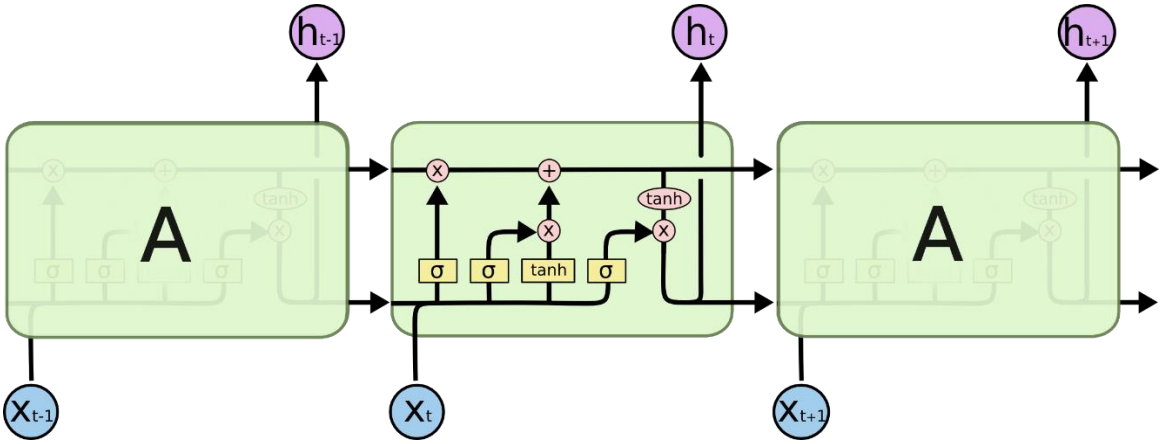
The RNN network I have implemented in 'Drum Notator' is a many-to-many network, which has a dynamic length (based on how many amplitude peaks in the audio there are). TensorFlow does not allow dynamic input/output shapes and therefore implements dynamic RNNs by inputting a certain number of timestamps (in this case each peak) at once, masking timestamps that do not have data at the start or end of a sequence. If the total number of timestamps exceeds this fixed size, then the network is run multiple times over different sets of timestamps. TensorFlow allows the functionality to only reset states when instructed, allowing dynamic length, and to return every timestamp's output, rather than just the last one, allowing many-to-many, rather than just many-to-one outputs.

Exploding / Vanishing Gradient

When training a neural network, 'loss' is aimed to be minimised to increase accuracy. The 'loss gradient' refers to the gradient of the function of loss over time in training. An 'exploding' gradient is therefore a gradient that increases over time (which is the opposite intention), and a 'vanishing' gradient is one that quickly approaches 0, limiting the maximum accuracy of the model (Geeks for Geeks, 2021). RNN networks can experience both these outcomes due to a high model complexity, and because conventional RNN networks take in the output of the last iteration, limiting their memory to the short term.

LSTM Networks

Long-Short-Term-Memory (LSTM) networks are RNNs that attempt to fix the exploding and vanishing gradient problems by allowing the cell control over what it remembers and what it forgets. This allows for memory to be retained for longer. As shown below, LSTM networks include more hidden layers which weigh inputs, outputs and carries, that enables it to learn what to remember and forget, on top of how to convert inputs and carries to appropriate outputs.



LSTM structure. Image cropped from (Jha, Long Short Term Memory [Image], 2016)

Unfortunately, LSTMs do not completely solve these gradient problems, however, reduce its effects (Geeks for Geeks, 2021). Furthermore, LSTMs are very susceptible to over fitting, as they can potentially memorise entire training dataset sequences, rather than adaptively solve problems. Dropouts can be used to help prevent this, which remove a certain random percentage of data to force the model to train in different ways.

7. References

Ando, K., Takamaeda-Yamazaki, S., Ikebe, M., Asai, T., & Motomura, M. (2016). The structure of the fully-connected layer (a) (b) and the data dependency (c) [Image]. *In Circuits and Systems, 08*, 149-170. Retrieved from https://www.researchgate.net/publication/318025612/figure/fig2/AS:510817424031744@1498799772936/The-structure-of-the-fully-connected-layer-a-b-and-the-data-dependency-c.png

Cornelisse, D. (2018). The filter slides over the input and performs its output on the new layer [Image]. *In An intuitive guide to Convolutional Neural Networks*. Retrieved from https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/

Geeks for Geeks. (2021). Understanding of LSTM Networks. Retrieved May 22, 2022, from https://www.geeksforgeeks.org/understanding-of-lstm-networks/

Google Developers. (2021). *Machine Learning Glossary*. Retrieved March 18, 2022, from https://developers.google.com/machine-learning/glossary

Google Developers. (2021). Pooling Convolution [Image]. *In Machine Learning Glossary*. Retrieved from https://developers.google.com/machine-learning/glossary/images/PoolingConvolution.svg

Jha, D. (2016). Long Short Term Memory [Image]. Retrieved from https://www.altoros.com/blog/introduction-to-neural-networks-and-metaframeworks-with-tensorflow/

Jha, D. (2016). Recurrent Neural Networks [Image]. Retrieved from https://www.altoros.com/blog/introduction-to-neural-networks-and-metaframeworks-with-tensorflow/

Richards, G. (2008). *The logistic curve* [Image]. Wikimedia Commons. Retrieved from https://en.wikipedia.org/wiki/Sigmoid_function#/media/File:Logistic-curve.svg

Roberts, L. (2020). Understanding the Mel Spectrogram. Retrieved May 15, 2022, from https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53