This document isn't meant for the general audience, it goes into a lot of detail of the process itself, however, it's not refined such that it's easily understandable. If you're looking for an easy to understand guide, there isn't any for now.

Part I
Preface

1 Define Difficulty

What makes a map difficult, what is a difficult map? Could it be the following? The map was difficult because of ...

1. Failing
2. Combo Breaks
3. High Stamina Requirement
4. Low Accuracy

We discuss all of these scenarios and we will choose one to tackle, possibly integrate the other options into our calculations in the future.

Failing The most significant way that we can readily control if players fail is via **Health Drain** in which most VSRGs will implement. However, this value is inconsistent and will not provide useful information on higher **Health Drain** values due to lack of players passing certain maps.

Combo Breaks Combo Breaks analysis is another method that isn't consistent, whereby chokes can be random, creating too much noise on higher skill plays. Combo breaks mainly can only determine the **hardest** points on the map, it doesn't depict a difficulty cure.

High Stamina Requirements While stamina is a good way to look at difficulty, it can readily be derived from accuracy, which is conveniently what we'll be looking at next

Low Accuracy This is the best way to look at difficulty, because not only it gives us a figure, it tells us the story and correlation between **accuracy** and **patterning**. This will be the main focus of the document.

1.1 Comparing Accuracy in Replays to Patterns

Details aside, how can we describe the impact of patterns on replays, what story does the replay tell us about the pattern?

Consider this...

1. $Player_1$ plays $Pattern_A$ and $Pattern_B$
2. $Player_1$ achieves $Accuracy_A > Accuracy_B$
3. Considering $Accuracy_A$ and $Accuracy_B$ are independent events
4. We deduce $Pattern_A < Pattern_B$ in difficulty

It is a simple idea to pitch, but we will have to dive into more details on how we can "teach" a machine this concept and "learn" from it!

2 Machine Learning

A tangent from this article, I will briefly talk about what is machine learning.

Making a Prediction Machine Learning is all about making a prediction, by looking at already curated data. *We show the someone 100 pictures of rabbits and explained to them that they are rabbits, can they tell if the next picture of an animal is a rabbit?*

Same idea we have here, we show the machine hundred of beatmaps, we tell the machine how difficult all of them are (according to score regression). Can the machine predict the difficulty of the next one? The answer is yes, but it won't be accurate!

2.1 Valid input

Think about a neural network, you've most likely seen one, it's circle(s) connected to more circle(s), in the end, there will be circle(s) that tell you something. Each circle represents a **neuron**.

Each neuron holds a numeric value, we need to put a value in each of these neurons. Ignoring Timing Points and Scroll Speed Changes, how do we squeeze a beatmap into these neurons?

Hit Object Neurons Remember that a neuron will only work with numeric values, what do you want to have in the neuron that represents a Hit Object? Putting either the **offset** or **column** will not work because both are vital!

Column Neurons It is a possible idea, to have a neuron for **each millisecond**, then put **column value** on those neurons that match the **offset**. We run into a glaring issue where input neurons will stretch to the **hundred thousands**, computational power required on this scale would be too high. If we decide to **bin** the offsets (binning is the act of grouping things together to reduce the number of fields) to **100ms** steps, it might prove to be too inaccurate and prone to abuse.

2.1.1 Subdivide the Issue

Does it have to be the whole map?

It doesn't! That's what we are doing for this research, we don't look at the whole map, we look at parts of it, the inner mechanisms, the **patterns**! Sure, even though the whole map must be present to calculate difficulty but consider this:

For each note in the map, the player will have to play **a pattern** (the input) and produce **a feedback** (the output). If we can parse a pattern and get an input, we should be able to teach the machine the expected output, and predict with it afterwards.

Patterns to Input In the following sections we will be discussing how we can task the machine to learn what patterns are easy and which are harder

Output to a Single Rating We can get output for each specific pattern, but how do we represent the difficulty as **one number?** Do we rate it by the average, maximum or median? We will discuss those later.

2.2 Expected output

Just like the example in **valid inputs**, we have to tell the machine that **that picture is of a cat**, else it doesn't know what is and what isn't.

In this case, we can relate back to *Accuracy* (from the player), where it is the most reliable source of difficulty rating as explained earlier.

One good source of this data is to grab replays (not scores!), because we can then analyse small parts of it in order to match with the input.

2.3 Differing and Multiple Accuracies

Multiple Players There will be multiple replays, there isn't a replay that is all-encompassing (it'll be too biased!). So we need to create a middle-ground for these data.

Different Players Not everyone will perform similarly for every beatmap, there will be discrepancies. We can't just take the accuracy as a raw value and run a function through them, we will have to make use of relative accuracies instead of raw accuracies before finding the "ideal" replay.

2.4 Differing Keys

Is it possible to mash all keys together, to form a model that works for 9 keys? I believe that it can be done, however, anything **beyond 7 keys** will be arguably bad in rating. This is mainly due to the lack of beatmaps in that category and also a lack of players. So how is it done?

Column to Action Is a small keyword I would use in my script to signify, map this column to the finger used in real life. To simply put, imagine if all columns were named by their most commonly pressed finger, that's the process I'm using to funnel all of them together.

2.5 Recap

To sum it up

1. Grab beatmap from server
2. Grab replay from server
3. Convert beatmap into simpler bite-sized patterns
4. Convert replay to a list of accuracies for each note
5. For each pattern, there's an expected accuracy
6. Teach that concept to the machine and create a model
7. Predict accuracies with the model
8. Give the map a rating according to the expected accuracies

Part II
Process

3 Grabbing Beatmaps

3.1 Beatmap Downloading

In this study we will only look at maps where $StarRating \geq 5.0$. Any maps below this $StarRating$ will usually only have **SS** scores, which will prove to be redundant in analysis.

osu!API GET beatmaps Using this, we can find out the IDs of the maps to download.

3.1.1 Downloading through web crawling

Using python, we can download *.osu* (osu! difficulty file extension) using the format below with authentication:

$$https://osu.ppy.sh/osu/ < beatmap\_id >$$

We save all of these using $< beatmap\_id >$ *.osu* file naming system.

All *.osu* will be converted to *.acd*. The action format is the same idea, where we have a list of offset and action.

4 Grabbing Replays

4.1 Replay Downloading

We need to look at *replays* as a mean to find out how well the player does (in other words, $difficulty$ of the map)

osu!API GET replay The API provides us with the replay file itself. Not going into detail, we are able to extract all key taps (including releases) of the player during the play.

These files are not saved, instead they are instantly decoded into the following format.

4.2 Replay Decoding

The format we get from running the python code goes as follows

$$action_{replay} \coloneqq \{(offset_1, action_1), (offset_2, action_2), ..., (offset_n, action_n)\}$$

Whereby,

$$n \in \{-9, -8, ..., -2, -1, 1, 2, ..., 8, 9\}$$

$offset$ is when the $action$ happens. For $action$, $-n$ means the key **n** is released, n means the key **n** is pressed.

We will save this data in a file with $< beatmap\_id >$ *.acr* extension.

5 Difficulty from Beatmaps

We turn our attention to how we can figure out difficulty from the map itself, the expected output we want would be:

$$difficulty \coloneqq \{(offset_1, difficulty_1), (offset_2, difficulty_2), ..., (offset_n, difficulty_n)\}$$

Whereby we estimate difficulty at offset **n** from the map itself:

$$difficulty_n \approx model\left(reading_n, \sum_{k=1}^{keys}(strain_k), ...\right)$$

There are more factors (denoted by ...) that contribute to difficulty, but we will regard them as noise in this research and fine tune this equation later.

Reading This denotes how hard is it to read all the patterns on the screen. We can draw similarities between this and density, however density focuses a lot more on its previous and future surroundings where reading looks at the future ones only.

Density This focuses on the **imminent** density of the offset. contrary to strain, it disregards the global trends of patterns. We will not use this in our network as strain does a better job in calculation.

Strain This is reliant on *density* whereby continuous high values of *density* will result in a high *strain*. This has an additional hyperparameter, *decay*, where it denotes how fast the player can recover from $strain_n$. Finger *strain* on the same hand will likely affect the other *strain* values of the other fingers.

5.1 Note Type Weights

This will define the **weightages** of each note type.

$weight_{NN}$ defines for normal notes

$weight_{LNh}$ defines for long notes heads

$weight_{LNt}$ defines for long notes tails

$weight_{SSh}$ defines for **strain shift** for hands **(explained later)**

$weight_{SSb}$ defines for **strain shift** for body **(explained later)**

5.2 Reading

$$reading_{(n,n+\theta)} \coloneqq count(NN), count(LNh), count(LNt) = \{n \leq offset \leq (n+\theta)\}$$

Where,

n is the initial offset

θ is the hyperparameter for length. We will not take into consideration the length of $note_{long}$

5.3 Density

We will look into density before strain as it's derived from this.

Considering the notes on the k column

$$\{..., n-2, n-1, n, n+1, n+2, ...\}$$

$$\Delta_{nx}^{k} = \frac{1}{|n-x|}$$

$$density_{n}^{k} = \sum_{N=n-\sigma}^{n+\sigma} \left(\Delta_{nN}^{k}\right)$$

So for $\sigma = 2$ and

$$column_{k} := \{a, b, n, d, e\}$$

$$density_{n}^{k} = \Delta_{na}^{k} + \Delta_{nb}^{k} + \Delta_{nd}^{k} + \Delta_{ne}^{k}$$

$$density_{n}^{k} = \frac{1}{|n-a|} + \frac{1}{|n-b|} + \frac{1}{|n-d|} + \frac{1}{|n-e|}$$

Δ_{nx}^{k} will be the the inverse of the (ms) distance between notes n and x on column k. Notes that are further away will be penalized more heavily.

σ defines the range, front and back of the search. Higher sigma may prove to be useless with further Δ_{nx}^{k} being too small.

5.4 Strain

This will work in relationship with *density*, whereby a *strain* is a cumulative function of *density* with a **linear decay function**.

Notes:

1. Better players have **higher decay gradients**
2. If $decay > density$, *strain* will **decrease**
3. If $decay < density$, *strain* will **increase**
4. There will be a point where *strain* is high enough to affect physical performance, indirectly affecting accuracy.

5.4.1 Strain Shift

Strain will not only affect one finger, it will affect the hand and both after time, just on a smaller scale

Hand We will denote the strain shift hyperparameter of one finger to another on the same hand to be SS_H

Body Likewise, for body, we will denote as SS_B

5.4.2 Strain Example

Consider the case, without **Strain Shift**

$$Where, weight_{NN} = 1, \sigma = 2$$

2500	0	0	0		0.022	0.016
2000	$weight_{NN}$	0	0	0.003	0.022	0.017
1500	$weight_{NN}$	0	0	0.005	0.019	0.015
1000	$weight_{NN}$	0	0	0.006	0.014	0.011
500	$weight_{NN}$	0	0	0.005	0.008	0.006
0	$weight_{NN}$	0	0	0.003	0.003	0.002
-500	0	0	0		0	0
-1000	0	0	0		0	0
Offset(ms)	k=1	k=2	k=3	$\approx Density$	Strain (dec=0)	Strain (dec=0.001)

Consider the case, with **Strain Shift**

2500	0	0	0
2000	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
1500	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
1000	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
500	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
0	$weight_{NN}$	$weight_{SSh}$	$weight_{SSb}$
-500	0	0	0
-1000	0	0	0
Offset(ms)	k=1	k=2	k=3

It's hard to include the calculations in the table, so we'll look at $density_{(1,1000)}$.

$$density^1_{1000} = (\Delta^1_{(1000,0)}) + (\Delta^1_{(1000,500)}) + (\Delta^1_{(1000,1500)}) + (\Delta^1_{(1000,2000)})$$

$$density^1_{1000} = \frac{1}{1000} + \frac{1}{500} + \frac{1}{500} + \frac{1}{1000} = 0.006$$

$$density^2_{1000} = (\Delta^2_{(1000,0)}) + (\Delta^2_{(1000,500)}) + (\Delta^2_{(1000,1500)}) + (\Delta^2_{(1000,2000)})$$

$$density^2_{1000} = (\frac{weight_{SSh}}{1000}) + (\frac{weight_{SSh}}{500}) + (\frac{weight_{SSh}}{500}) + (\frac{weight_{SSh}}{1000}) = \frac{3 * weight_{SSh}}{250}$$

$$density^3_{1000} = \frac{3 * weight_{SSb}}{250}$$

$$density_{1000} := density^1_{1000}, density^2_{1000}, density^3_{1000} = \{0.006, \frac{3 * weight_{SSh}}{250}, \frac{3 * weight_{SSb}}{250}\}$$

5.5 Density Generalization

In the case where we want to find $density_n$, where, n is the offset index, k is key count.

$$\begin{bmatrix} weight_{(n+\sigma,1)} & weight_{(n+\sigma,2)} & \dots & weight_{(n+\sigma,k)} \\ \vdots & \vdots & \iddots & \vdots \\ weight_{(n+1,1)} & weight_{(n+1,2)} & \dots & weight_{(n+1,k)} \\ weight_{(n,1)} & weight_{(n,2)} & \dots & weight_{(n,k)} \\ weight_{(n-1,1)} & weight_{(n-1,2)} & \dots & weight_{(n-1,k)} \\ \vdots & \vdots & \ddots & \vdots \\ weight_{(n-\sigma,1)} & weight_{(n-\sigma,2)} & \dots & weight_{(n-\sigma,k)} \end{bmatrix}$$

$$*$$

$$\begin{bmatrix} offset_{n+\sigma} & \dots & offset_{n+1} & offset_n & offset_{n-1} & \dots & offset_{n-\sigma} \end{bmatrix}$$

$$=$$

$$\begin{bmatrix} density_{n+\sigma} & \dots & density_{n+1} & density_n & density_{n-1} & \dots & density_{n-\sigma} \end{bmatrix}$$

$$density_n \coloneqq \begin{bmatrix} density_{n+\sigma} & \dots & density_{n+1} & density_n & density_{n-1} & \dots & density_{n-\sigma} \end{bmatrix}$$

From here, we can calculate the strain by running the through a python code.

5.6 Allocating Notes to Fingers

We cannot assume that $column_1$ where $keys = 4$ is the same as $column_1$ where $keys = 7$. This is due to how **different fingers interact with the same column**.

As to counter this, we need to find out the **most common set-up** for players.

key	LP	LR	LM	LI	S	RI	RM	RR	RP
4			1	2		3	4		
5			1	2	3	4	5		
6		1	2	3		4	5	6	
7		1	2	3	4	5	6	7	
8	1	2	3	4		5	6	7	8
8S	1	2	3	4	5	6	7	8	
9	1	2	3	4	5	6	7	8	9

L represents left, **R** is right, the second letter will be th e name for the fingers (LP: Left Pinky, LR: Left Ring and so on...)

This allocation will give us a consistent result for all beatmaps, so $key = 1$ will always mean **Left Pinky**, $key = 2$ for **Left Ring**, and so on...

5.6.1 8 Key Scratch Bias

The issue with 8 Key maps is how maps will usually have a *scratch column*, this will create a setup that **excludes a pinky but includes the thumb**. Due to this, we will shift the configuration to left pinky and right thumb, excluding the right pinky. See the above **8S**

5.7 Assigning Hyperparameters

In this section alone, we have used quite a few hyperparameters. To recap:

(Reading) θ is the hyperparameter for reading length.

(Density) σ defines the range, front and back of the density search. Higher sigma may prove to be useless with further Δ_{nx}^{k} being too small.

(Density) $weight_{NN}$ defines for normal notes

(Density) $weight_{LNh}$ defines for long notes heads

(Density) $weight_{LNt}$ defines for long notes tails

(Density) $weight_{SSh}$ defines for **strain shift** for hands

(Density) $weight_{SSb}$ defines for **strain shift** for body

Now what we need to do is to assign a reasonable values to these, and run the results to find our:

$$difficulty \coloneqq \{(offset_1, difficulty_1), (offset_2, difficulty_2), ..., (offset_n, difficulty_n)\}$$

Whereby we estimate difficulty at offset **n** from the map itself:

$$difficulty_n \approx model\left(reading_n, \sum_{k=1}^{keys}(strain_k), ...\right)$$

We can further expand this to:

$$difficulty_n \approx model\left(count(NN), count(LNh), count(LNt), \sum_{k=1}^{keys}(strain_k)\right)$$

Where strain

5.7.1 Values of Hyperparameters

There will definitely be issues when it comes to assuming hyperparameters because of how it will affect accuracy of the model. However, as long as we **reasonably** assign them, most of the errors will be offset by the neural network learning. We just need to focus on if a certain value should be **larger/smaller** or **negative/positive**.

(Reading) θ 1000 (ms)

(Density) σ 2

(Density) $weight_{NN}$ 1

(Density) $weight_{LNh}$ 0.75 (we will follow Reading)

(Density) $weight_{LNt}$ 0.75 (we will follow Reading)

(Density) $weight_{SSh}$ 0.25

(Density) $weight_{SSb}$ 0.1

6 Difficulty from Replays

We are able to extract data from replays, however, they are not linked to the beatmaps themselves. This means that it only has data of what keys and when did the player press it, there's no data on accuracy achieved.

To recap, we managed to decode the replay sent into the following format:

$$action_{replay} \coloneqq \{(offset_1, action_1), (offset_2, action_2), ..., (offset_n, action_n)\}$$

Whereby,

$$n \in \{-9, -8, ..., -2, -1, 1, 2, ..., 8, 9\}$$

$offset$ is when the $action$ happens. For $action$, $-n$ means the key **n** is released, n means the key **n** is pressed. In this section, we will be discussing how we can make this a suitable output for the neural network to predict.

6.1 Mapping a Replay to Beatmap

In this, we match all similar actions in their respective columns.

There are a few things we need to take note of when matching:

1. Not all $action_{beatmap}$ will have a matching $action_{replay}$
2. We put the threshold of this matching as $100ms$, i.e. $action_{beatmap}$ that doesn't have any $action_{replay}$ within $100ms$ will be regarded as a miss.
3. The nearest $action_{replay}$ will match the $action_{beatmap}$, not the earliest one.
4. We will deviate on how osu! calculate accuracy due to the above pointers, this allows us to calculate on a more common basis.

We will expect the output of:

$$deviation \coloneqq \{(offset_1, deviation_1), (offset_2, deviation_2), ..., (offset_n, deviation_n)\}$$

Where:

$$n = length(action_{beatmap})$$

And if there's no match, $deviation > 100$, this is to allow us to understand that it's a **miss** instead of a $100ms$ hit.

This doesn't proportionally represent accuracy (which will be easier to understand), as its lower value represents a better judgement, so we will adjust to the following:

$$accuracy \coloneqq \{(offset_1, accuracy_1), (offset_2, accuracy_2), ..., (offset_n, accuracy_n)\}$$

Where:

$$accuracy_n = 100 - deviation_n$$

Therefore, a **miss** would simply just be $accuracy = 0$ instead.

So accuracy will only span:

$$(Miss)0 \leq accuracy_n \leq (Perfect)1$$

$$where, deviation_n \in [0, 1, 2, ..., 99, 100]$$

6.2 Replay Soloing

As discussed in the preface, we need to resolve two issues. **Multiple Replays** and **Multiple Players**. We will expect an output similar to a replay, this is where the first major assumption kicks in.

6.2.1 Assumption of the Top 50

In this, we assume that if we took the **median** all top 50 replays, we will end up with a replay that is all-encompassing.

This leads some problems:

Top Player Bias The neural network will perform worse on easier maps $\approx 3.0S.R.$ due to the median in easier maps being too consistently perfect, this leads to amplification of noise (in this case, chokes).

Population Interaction This assumption will only hold if the beatmaps are old enough such that most of the general playerbase has played it, else it doesn't represent the population well enough due to low participation

Population Decay/Improvement The beatmaps we check must be ranked within close proximity with each other, this is to avoid the Top 50 median from adjusting too much

6.2.2 Assumption of the Player

In this, instead of looking at it **per beatmap**, we will do it **per player**. This is much more consistent in data, however it's consistent **for that player only**.

In this method, we grab the all replays that **Player X** has played and compare them with each other. However this still leads to assumptions:

1. The player played all the beatmaps at a similar time, or the player never improved
2. The player is representative of the community (bias)

The problem of the machine being **biased** to only the player can be fixed if we ran an averaging function on the output with hundreds of other players.

However, due to how osu! sends API scores, I couldn't get more than **50** scores **per player**. This means that there will be no data for these replays.

6.3 Choosing an Assumption

Despite this, we will work with **Assumption of the Top 50** as it's reasonable enough, and it's easier.

This means that the median of all replays will be saved in a different data file with $< beatmap\_id > .acrv$ extension. *(v represents virtual)*

6.4 Virtual Player/Replay

The median of the top 50 creates a **virtual replay**, where we expect it to be a **good enough** representation of a **virtual player**. We can now pivot from this player as we calculate difficulty!

6.5 Smoothing

As expected, the median gathered will not be a smooth graph, we will smooth out with an **moving aggregation of its mean with a window of 30**. This means that it'll grab 30 data points, front and back, and calculate its median, creating a new series/vector.

As player's deviations usually will not be consistent throughout, this will help hammer down large errors, while also affecting its neighbouring data points. In turn, this aids the machine to learn that the error may not only be the issue of that one note, but instead a group of it.

7 Creating a Neural Network

Before we dive into creating a neural network to finalize the calculation, we need to define specifically these few things:

1. Input
2. Output
3. Layers

Input Our input will be a vector containing information about 12 parameters.

#	name	description
1	LP	Left Pinky
2	LR	Left Ring
3	LM	Left Middle
4	LI	Left Index
5	S	Spacebar
6	RI	Right Index
7	RM	Right Middle
8	RR	Right Ring
9	RP	Right Pinky
10	NN(count)	Normal Note (count)
11	LNH(count)	Long Note Head (count)
12	LNT(count)	Long Note Tail (count)

Output The output will be the expected *accuracy*, which is gathered from the replays.

#	name	description
1	roll	Rolling average (window=30) of the replays' Medians

Layers The amount and neurons of layers we use will be a hyperparameter. We will assign a value for those later, and talk more about this in the next part, we will firstly look into **bad data**

7.1 Bad Data Filtering

One of the biggest problems in the model is how we didn't take into consideration Scroll Speed. This stems from ignoring them in the difficulty calculation *from beatmaps*.

7.1.1 Scroll Speed Changes

It's hard to tell specifically which data is affected by the scroll speed change, but it's easy to tell what maps have scroll speed changes. However, by totally ignoring any maps with a slither of scroll speed changes, we will be left with a small group of **hard maps**. This is largely due to the nature that harder maps are frequently paired with them, so we need to find a filter.

Scroll Speed Manipulation Selection This is a subjective issue to tackle. I personally have tried doing an automated rejection of maps that have more than 1 SV or BPM value, however that proved to fail due to maps having intentional scroll speed changes, despite not oriented to be difficult in that aspect.

We will source out these maps manually, it is not a hard task, so it will be worth the effort.

7.2 Drafting the Model

Firstly, we need to identify the nature of this issue. This falls under the category of **Regression** as we are dealing with a range of values, instead of a **yes** or **no**, which is instead labelled **Classification**.

The main problem we will face is scoring. We can label how many **yes** or **no** predictions are *false positives*, but we require another method to score how well we did by prediction.

7.3 Custom Scoring

We will use a custom scoring to find out if the model has done well. It's a simple function we are going to use, which is to find the **average absolute difference** between the expected graph and predicted graph. In mathematical terms:

$$\frac{\sum(|prediction - actual|)}{length(prediction)} = score$$

Where, the score is better if it's closer to 0. We also calculate the *standard deviation* of the absolutes as an extra parameter to reference.

Part III
Refinement

8 Refining Neural Network

In this section, we will be talking about how the neural network goes through change to score best.

8.1 Initial Draft

For this neural network, we will firstly start off with the following code. A simple neural network, with 2 layers, the input, and the output.

```
def model_c():

    model = keras.models.Sequential()
    model.add(keras.layers.Dense(<neurons>, input_shape=(12,), \
                kernel_initializer='normal', activation='relu'))
    model.add(keras.layers.Dense(1, kernel_initializer='normal'))
    model.compile(loss='mean_squared_error', optimizer='adam')

    return model
```

input_shape This defines how many dimensions we have the input as, we have 12 parameters, hence 12.

kernal_initializer This defines how the weights are randomly allocated before the learning process. For this we just use the normal.

activation We will simply be using **ReLU** (rectified linear unit) as our activation function. This simply maps any positive values in the neuron to a linear function, anything non-positive is mapped to 0.

loss The loss describes how far the machine is predicting all values correctly. In this case, we will use mean squared error as our minimum loss target as we are dealing with a *regression* problem.

optimizer It's hard to describe what is **Adam** (adaptive moment estimation), however, it is described to be "better" than **SGD** (Stochastic Gradient Descent). More info here: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

8.1.1 Layer Improvements

We will keep it short on the guess and check for neuron layers, here are the results and improvements done.

Parameters						Loss	
#	L_1	L_2	L_3	Epochs	Batch Size	Mean	STDEV
1	96	48	24	10	500	1.80	0.60
Drop Batch Size to 200							
2	96	48	24	10	200	1.81	0.58
Half Neurons							
3	48	24	12	10	200	1.81	0.61
Add Reg L1 (0.01) on L_2 and L_3							
4	96	48	24	10	200	1.67	0.67
5		$L1_{0.01}$	$L1_{0.01}$				
Add Reg L1 (0.02) on L_2 and L_3							
6	96	48	24	10	200	1.73	0.63
		$L1_{0.02}$	$L1_{0.02}$				
Add Reg L2 (0.01) on L_2 and L_3							
7	96	48	24	10	200	1.72	0.64
		$L2_{0.01}$	$L2_{0.01}$				
Add Dropout (0.25) after L_2 and L_3							
8	96	48	24	10	200	1.73	0.63
		$L1_{0.01}$	$L1_{0.01}$				
		$DO_{0.25}$	$L2_{0.25}$				
Revert to 7, increased Epoch and Decreased Batch Size							
9	96	48	24	50	50	1.69	0.56
		$L2_{0.01}$	$L2_{0.01}$				
Increase Reg L2 to 0.03							
10	96	48	24	50	50	1.81	0.57
		$L2_{0.03}$	$L2_{0.03}$				
Revert to 7, doubled neurons							
11	192	96	48	50	50	1.75	0.57
		$L2_{0.03}$	$L2_{0.03}$				

In the end of the experimenting, we will settle on **Model 9**

Part IV
Results

9 Results

We will now test the model with high star rated maps

```
scor_medn_metadata
1.15_06.70_typeMARS - Triumph & Regret (Regret)
1.80_06.75_ginkiha - Anemoi (Daybreak)
1.43_06.85_ginkiha - Borealis (GRAVITY)
1.22_07.04_xi - over the top (Extra)
0.97_07.04_Hermit - Dysnomia (Tidek's 4K Monochrome)
1.00_07.05_Imperial Circus Dead Decadence - Uta (Tragic Ends)
2.15_07.19_Utsu-P feat.Kagamine Rin - Tokyo Teddy Bear (SHD)
1.43_07.47_Team Grimoire - G1ll35 d3 R415 (Shana's Extra)
1.19_07.52_MAZARE - Mazare Party (Ash's 4K Extra)
1.54_07.55_Team Grimoire - G1ll35 d3 R415 (L45T C4LL)
0.87_07.55_Colorful Sounds Port - ETERNAL DRAIN (Black Another)
1.18_07.63_TeamGrimoire+amaneko - croiX (GRAVITY)
1.44_07.64_Helblinde - Memoria (Original Mix) (Memories)
1.20_07.64_Hypernite Industries - Speedcore 300 (Extra)
1.03_07.71_kamome sano - archive::zip (GRAVITY)
1.10_07.77_ZOGRAPHOS (Yu_Asahina+Yamajet) - Verse IV (INFINITE)
1.45_07.78_Team Grimoire - C18H27NO3(extend) (4K Capsaicin)
0.75_07.86_Kobaryo - Dotabata Animation [feat. t+pazolite] (Ultra...
0.99_07.88_Camellia as "Bang Riot" - Blastix Riotz (Jinjin's INFI...
1.29_07.98_Kurokotei - Galaxy Collapse (Cataclysmic Hypernova)
1.28_08.02_Team Grimoire - Sheriruth (Arzenvald's EXtinction)
0.64_08.02_Colorful Sounds Port - ETERNAL DRAIN (Eternal)
0.67_08.07_UNDEAD CORPORATION - The Empress scream off ver (Zenx'...
1.02_08.08_Chroma - I (Chicken AI's Maximum)
1.60_08.10_Kaneko Chiharu - Zettai Reido (GRAVITY)
2.57_08.10_S-C-U feat. Qrispy Joybox - anemone (Kawa & Julie's 7K...
0.98_08.12_RoughSketch feat.Aikapin - Grimm (Kobaryo's FTN-Remix)...
1.10_08.23_AAAA - Hoshi o Kakeru Adventure ~ we are forever frien...
1.12_08.26_Camellia as "Bang Riot" - Blastix Riotz (Shirou's EXTR...
2.51_08.45_Seiryu - Time to Air (Bruce's 7K Another)
1.10_08.51_P*Light feat. mow*2 - Homeneko*Sensation (8K Lv.12)
1.99_08.54_S-C-U feat. Qrispy Joybox - anemone (Kawa & Julie's 8K...
1.05_08.64_Kaneko Chiharu - iLLness LiLin (HEAVENLY)
1.59_08.73_Amane - Space Time (Amane Hardcore Remix) (7K Lv.40)
0.97_08.73_Ryu* - Yukizukiyo (victorica's 8K Lv.12)
2.42_08.74_GReeeeN - Shinobi (7K Burst!)
0.77_08.75_xi - Double Helix (Nucleic)
1.63_08.76_Yooh - Ice Angel (Euphoria)
1.34_08.83_Ogura Yui - Baby Sweet Berry Love (3R2 Remix) (Sweetne...
1.64_08.87_Ryu* Vs. Sota - Go Beyond!! (7K Black Another)
1.63_08.91_void - Valedict (Black Another)
1.55_08.92_Cres - End Time (Epilogue)
1.10_08.92_High Speed Music Team Sharpnel - M.A.M.A. (ExTra)
0.72_08.97_XeoN - Xeus (LV.12 Leggendaria)
```

1.20_08.97_D(ABE3) - MANIERA (Collab Another)
1.20_08.98_orangentle / Yu_Asahina - HAELEQUIN (Extended ver.) (J...
1.25_09.01_penoreri - Everlasting Message (GRAVITY)
1.68_09.02_Gekikara Mania - Deublithick (Blocko's Extra)
1.30_09.02_Cres - End Time (Afterword)
1.03_09.02_Sisterz - Inverse World (Universe)
1.34_09.05_w_tre respect for AT&HU - Schur's Theorem (Black Anoth...
1.63_09.07_Hermit - Dysnomia (8K Melodie de tristesse)
1.37_09.07_Sharlo & yealina - Kakushigoto (Rumi's 7K MX)
1.66_09.08_Shoujo - Reminiscing (Black Another)
3.66_09.08_xi - Happy End of the World (9K Collab Insane)
1.25_09.09_BlackY - Harpuia (INFINITE)
1.59_09.09_penoreri - Preserved Valkyria (GRAVITY)
1.10_09.10_Yooh - LiFE Garden (Extended Mix) (Eutopia)
1.33_09.10_kamome sano - archive::zip (GRAVITY)
1.12_09.12_Soleily - Violet Soul (D's Extra)
1.67_09.12_LeaF - LeaF Style Super*Shredder (SC)
1.61_09.14_xi - Quietus Ray (Heaven)
1.51_09.15_DJ TOTTO VS TOTTO - Vajra (Emiria's 8K Leggendaria)
1.95_09.16_Hermit - Dysnomia (D's 8K Another)
1.58_09.18_MiddleIsland - Achromat (7K Black Another)
1.19_09.20_gmtn. (witch's slave) - furioso melodia (7K furioso ma...
1.58_09.22_DJ SHARPNEL - Touch the angel (K-ON!!)
0.81_09.23_MAZARE - Mazare Party (Ash's 6K Extra)
1.87_09.28_DJ Mashiro - Prismatic Lollipops (Lv.20)
0.95_09.28_Soleily - Renatus (Another)
1.21_09.29_BlackY - FLOWER -SPRING Long VER.- (As Flowers Bloom A...
1.81_09.30_Nekomata Master - Sennen no Kotowari (GRAVITY)
0.74_09.31_UNDEAD CORPORATION - The Empress scream off ver (Jepet...
1.37_09.36_Team Grimoire - C18H27NO3(extend) (7K Axities)
1.69_09.38_Camellia as "Bang Riot" - Blastix Riotz (GRAVITY)
1.65_09.39_Kucchi- - Remilia ~Kyuuketsuki no Tame no Kyousoukyoku...
1.80_09.51_Kucchi- - Remilia ~Kyuuketsuki no Tame no Kyousoukyoku...
1.26_09.52_Halozy - Kanshou no Matenrou (Eternity)
1.63_09.54_xi - Garyou Tensei (Million's 7K MX)
1.29_09.54_Mitsuyoshi Takenobu no Ani - Amphisbaena (Fallen Heave...
1.59_09.56_Chroma - Hoshi ga Furanai Machi (Meteor Shower // ppor...
1.72_09.58_LeaF - Alice in Misanthrope -Ensei Alice- (Alice in Wo...
1.33_09.58_Umeboshi Chazuke - Panic! Pop'n! Picnic! (Picnic!)
1.10_09.65_xi - Ascension to Heaven (Elysium)
1.57_09.65_Ayane - Endless Tears... (CrossOver)
1.07_09.65_CLIMAX of MAXX 360 - PARANOiA Revolution (Expert)
1.22_09.67_Jun Kuroda + AAAA - Ultimate Fate (The Apocalypse)
1.35_09.69_Warak - REANIMATE (Reanimated obj. Kamikaze)
1.29_09.69_Ice - citanLu (pporse's Lunatic)
1.28_09.72_Yooh - FIRE FIRE -DARK BLAZE REMIX- (GRAVITY)
1.55_09.75_xi - Happy End of the World (ajee's 5K Armageddon)
1.74_09.79_m108 - * Crow Solace * (KK's 7K Extra)
1.22_09.79_Cardboard Box - The Limit Does Not Exist (Extra)
1.63_09.86_DragonForce - Symphony of the Night (Rhapsody of the W...
1.97_09.88_xi - F (X)
1.89_09.91_DJ Genki VS Camellia feat. moimoi - YELL! (Solitude)
0.94_09.93_Gekikara Mania - Deublithick (Ultra)

```
1.14_09.98_KRUX - Illusion of Inflict (7K Kruxified)
1.22_10.02_Toromaru - Enigma (GRAVITY)
1.46_10.02_sakuzyo - AXION (Ex-ray)
1.53_10.06_xi - Garyou Tensei (Million's 7K SC)
1.92_10.06_DJ Genki vs. Camellia feat. moimoi - Sunshine (7K Lumi...
1.44_10.10_m108 - * Crow Solace * (richard's 7K Soluis)
1.81_10.13_LeaF - LeaF Style Super*Shredder (SHD)
3.25_10.20_saradisk - 220 - Kumano (Kumano!! Kumano!! Kuma!!)
1.70_10.20_D(ABE3) - MANIERA (Masterpiece)
1.23_10.23_xi - ANiMA (Pew's 7K Lv.36)
1.24_10.28_Cardboard Box - The Limit Does Not Exist (Infinity)
1.30_10.33_Umeboshi Chazuke - Panic! Pop'n! Picnic! (Kawawa's Ex7...
1.29_10.34_void - Just Hold on (To All Fighters) (Blocko & Soul's...
0.93_10.36_DETRO - volcanic (Pyroclasm)
1.89_10.42_kamome sano - archive::zip (GRAVITY) <Fresh Chicken>
3.17_10.49_senya - Mahou ga Umareta Hi (Elegance Lunatic)
1.43_10.59_technoplanet - Inscape (HEAVENLY)
1.44_10.60_Yu_Asahina - Ongaku -resolve- (Music)
1.21_10.87_void - Just Hold on (To All Fighters) (Resolve)
1.91_10.90_DragonForce - The Warrior Inside (6K Collab Guardian)
2.36_11.08_xi - Happy End of the World (9K Meteor Shower)
1.26_11.39_Doin - Further (Lulu's Go Beyond)
1.54_11.40_Lime - Smiling (Be Happy)
1.38_11.40_Gekikara Mania - Deublithick (Mania)
1.94_11.75_MAZARE - Mazare Party (JAKARE's 6K Extra)
2.28_11.86_HO-KAGO TEA TIME - GO! GO! MANIAC (TV Size) (K-ON!!)
2.23_11.91_OISHII - ONIGIRI FREEWAY (Unagi Onigiri "UHD")
```

There are more results in the GitHub Repo.

10 Conclusion

10.1 The model isn't reliable

While I think the model is good to a certain degree, I don't think it's stable enough compared to a non-black box model. In other words, I don't think this model is production ready for a few reasons:

1. The model doesn't calculate less popular key counts well
2. The model leans to the Top 50 population, so
   (a) The model is bad at calculating easy maps as there's little to no data with regards to deviation in those
   (b) The model is bad at matching situations where 2 maps are actually the same difficulty, but due to low interaction, the medians are vastly different.
   (c) The model doesn't have enough results for bad deviation situations.
   (d) The model seems to be reliably calculating how hard is it to get a very high score (i.e. Varying LN ends are hard to perfect, compared to hard jacks), however, most player's perception of difficulty rating doesn't fall in line with that.
3. The model has very strong bias towards maps that are popular, e.g. Triumph & Regret, which seems to be heavily affected by this issue
4. Even if the model works, it's hard to explain what happens in the model since the neural network acts like a black-box.

10.2 How to fine-tune the model

I think the main reason of this issue is the **Assumption of the Top 50**. In other words, we assumed that the Top 50 results' median can be treated like a player. However, this assumption proved to be very disastrous in terms of reliability.

1. One way to prevent this is to look at specific player's replays and do the same modelling again, however, we will pivot from the player's perspective.
2. We could have also, instead of looking at Top 50, we would focus on the a specific range of scores (which is painfully annoying to grab from the API), e.g. 750000 – 850000, in which will give us more information on bad deviations instead of extremely high scores with less bad deviations.
3. We could have dropped the idea of looking at key counts over 7 due to the small amount of data we would have gotten from them. This has created a lot of noise in the model.
4. We did not take into account in-depth reading difficulty (i.e. how hard is it to read a broken stair compared to a smooth one). If we could've calculated that one reliably, it would've smoothed out the LN map difficulties
5. If we were able to grab Double Time results, it'll double our data, and increased the pool of harder maps, this makes it significantly easier for the machine to rate them.

Part V
Annex

11 Scripts

In this, I'll be talking about the scripts I used. Note that this is for reference, for myself, in the future, so for anyone else reading it, it may be a bit confusing.

11.1 Parsing a beatmap

In the hierarchy, it'll look like this:

```
[get_osu_from_website.py]
bm_id    -[DL]-> osu

[osu_to_osus.py]
osu      -[Py]-> osuho + osutp + params
osutp    :                used to check for scroll speed changes
params   :                beatmap metadata

[osuho_to_acd.py]
osuho    -[Py]-> acd

[get_plyrid.py]
bm_id    -[API]> plyrid

[plyrid_to_acr.py]
plyrid   -[Py]-> acr

[ac_to_acrv.py]
acr+acd  -[Py]-> acrv

[ac_to_ppshift.py]
acrv+acd-[Py]-> ppshift
```

11.2 Feeding the map

```
[interface_neural_network.py]

    # This is the overarching public function
    def train(self, epochs: int, batch_size: int):
        self._load_training()
        self._train_model(epochs, batch_size)

    # This converts the ppshift into a usable pandas.Dataframe
    def _load_training(self):

        print("Merging " + str(len(self.training_ids)) + " files")
        print(self.training_ids)
```

```
        ppshift_list = []

        for bm_id in self.training_ids:
            ppshift_f = open(self.ppshift_dir + str(bm_id) + '.ppshift', 'r')
            # Important: The first 29 results has a 0 output, so we will cut
            # those results out with splicing.
            # This is due to the rolling Aggregation
            ppshift_str = ppshift_f.read().splitlines()[29:]
            ppshift_f.close()
            ppshift_list.extend(list(map(eval, ppshift_str)))

        self.training_df = \
        pandas.DataFrame(ppshift_list,\
                         columns=['OFF','LP','LR','LM','LI','S', \
                                  'RI','RM','RR','RP','NN','LNH','LNT','MED'])


    # This is the Keras Model itself, based on model 9 in Part III
    def _model_c(self):

        model = keras.Sequential()

        # We make this user-friendly to adjust
        self.layer_1_nrns = 96
        self.layer_2_nrns = 48
        self.layer_3_nrns = 24

        # Layer 1
        model.add(keras.layers.Dense(\
            self.layer_1_nrns, input_shape=(12,), \
            kernel_initializer='normal', \
            activation='relu'))

        # Layer 2
        if (self.layer_2_nrns != None):
            model.add(keras.layers.Dense(
                self.layer_2_nrns,\
                ernel_regularizer=keras.regularizers.l2(0.01)))

        # Layer 3
        if (self.layer_3_nrns != None):
            model.add(keras.layers.Dense(
                self.layer_3_nrns,\
                kernel_regularizer=keras.regularizers.l2(0.01)))

        # Output Layer
        model.add(keras.layers.Dense(1, kernel_initializer='normal'))

        model.compile(loss='mean_squared_error', optimizer='adam')

        return model

    # This feeds the machine
```

```
def _train_model(self, epochs: int, batch_size: int):

    # This is the dataframe, extracted from ppshift
    df = self.training_df

    # Shuffle
    df = df.sample(frac=1)
    ds_s = df.values

    in_ds_s = ds_s[:,1:13]
    out_ds_s = ds_s[:,13]

    model = self._model_c()
    model.fit(in_ds_s, out_ds_s, epochs=epochs, batch_size=batch_size)

    # Save model
    model.model.save(self.model_dir + self.model_name + '.hdf5')

    self.model = model
```

11.3 Testing the model

```
# This is the overarching public function
def test(self):
    self._load_model()
    self._test_model(self.testing_ids, 'results_tst')
    self._test_model(self.training_ids, 'results_trn')

# This simply gets the model
def _load_model(self):

    def dummy():
        return
    model = KerasRegressor(build_fn=dummy, epochs=1, batch_size=10, verbose=1)
    model.model = ld_mdl(self.model_dir + self.model_name + '.hdf5')

    self.model = model

# This will test the model and use a custom scoring method
def _test_model(self, bm_ids, log_file_name = 'results'):

    rating_list = []
    log = []

    for bm_id in bm_ids:
        ppshift_f = open(self.ppshift_dir + str(bm_id) + '.ppshift', 'r')
        # Important: The first 29 results has a 0 output, so we will cut
        # those results out with splicing.
        # This is due to the rolling Aggregation
        ppshift_str = ppshift_f.read().splitlines()[29:]
        ppshift_f.close()

        ppshift_df = pandas.DataFrame(list(map(eval, ppshift_str)))
```

```
        ds = ppshift_df.values

        in_ds = ds[:,1:13]
        out_ds = ds[:,[0,13]]

        # We predict the results here
        out_p = self.model.predict(in_ds, verbose=0)

        out_o = pandas.DataFrame(out_ds, columns=['offset','original'])
        out_p = pandas.DataFrame(out_p, columns=['pred'])

        # We join back with the original dataframe to compare
        out = out_o.join(out_p)

        # This is where the plots created is rated
        out['delta'] = out['pred'].subtract(out['original']).abs()
        log.append(str(out['delta'].mean()) + "\t" + \
                   get_beatmap_metadata.metadata_from_id(int(bm_id))

        rating_list.append(out['delta'].mean())

    # Append stats
    log.append("mean: " + str(statistics.mean(rating_list)))
    log.append("stdev: " + str(statistics.stdev(rating_list)))
    log.append("range: " + str(min(rating_list)) + ' - ' \
               + str(max(rating_list)))

    # Export to a .txt file
    log_f = open(self.plot_dir + log_file_name + ".txt", "w+")
    log_f.write('\n'.join(log))
```