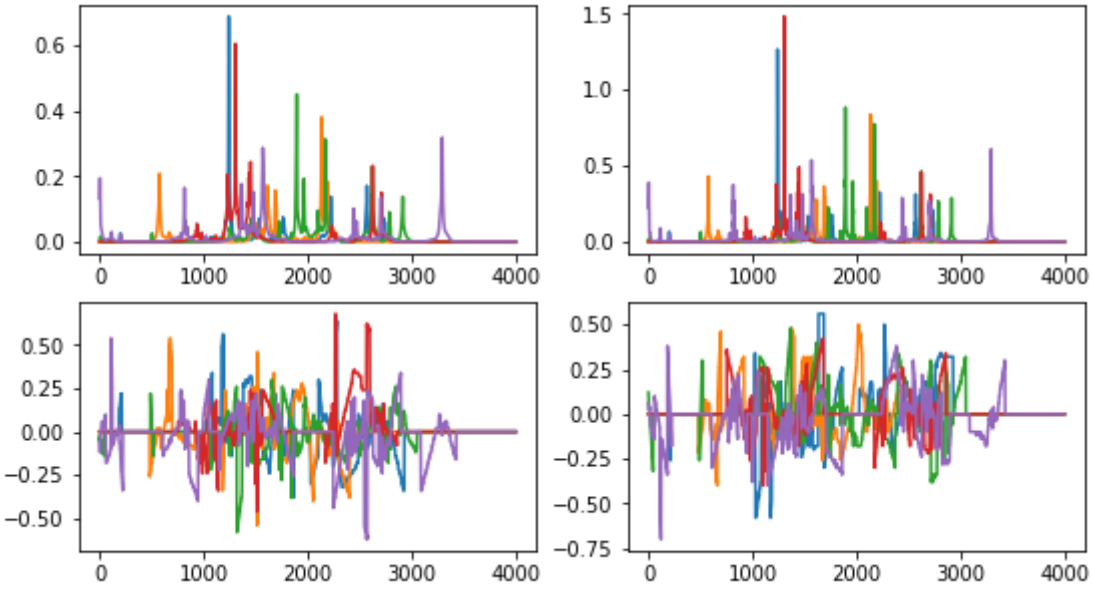
Question 1 - DATA VISUALISATION

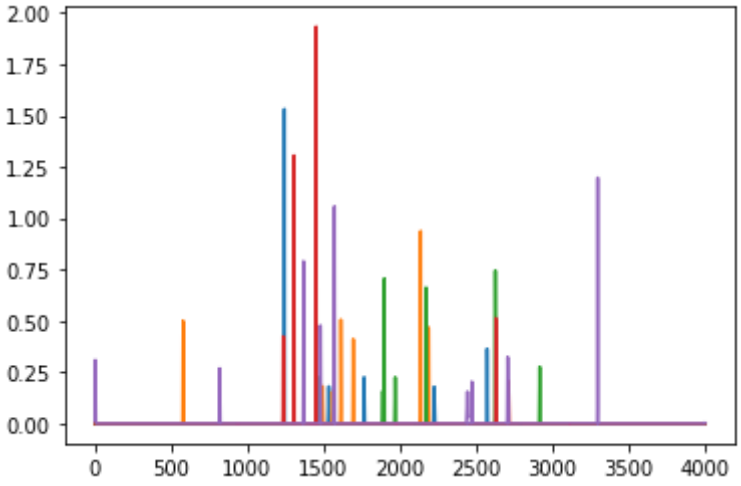
1 The Rage of Data

Here I took 5 random files and one random sample from each file, which gave me 5 sets of x and y points, I separated the four tensors in the Xs and plotted them each, and along with this, I plotted the y as well. Most of the values are just 0s.

The four tensors are each plotted on an x-axis of 4000 points



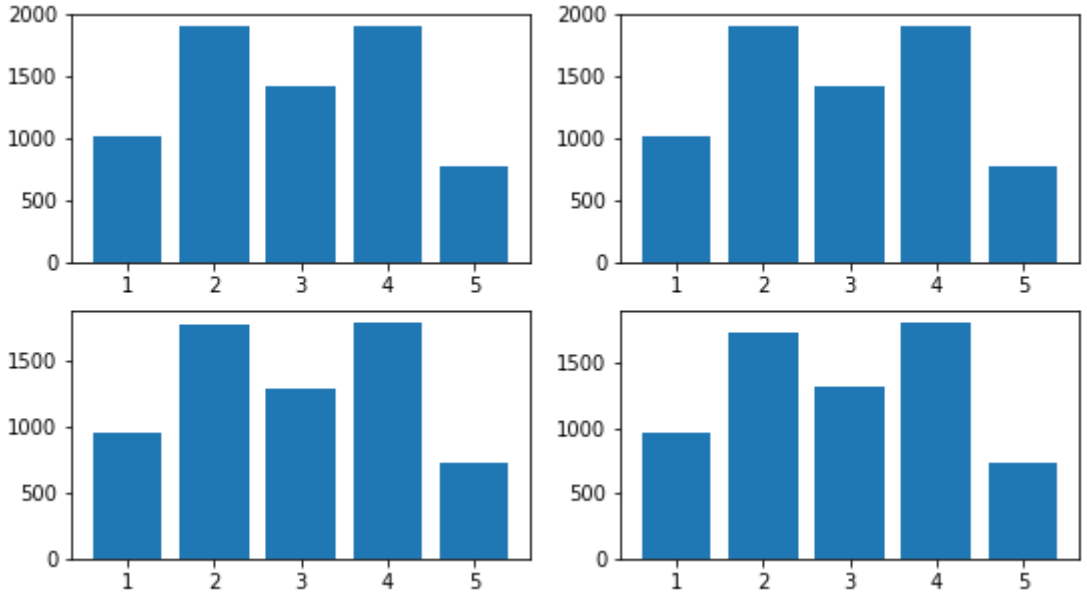
The y tensor of 4000 points plotted on an axis of 4000 points



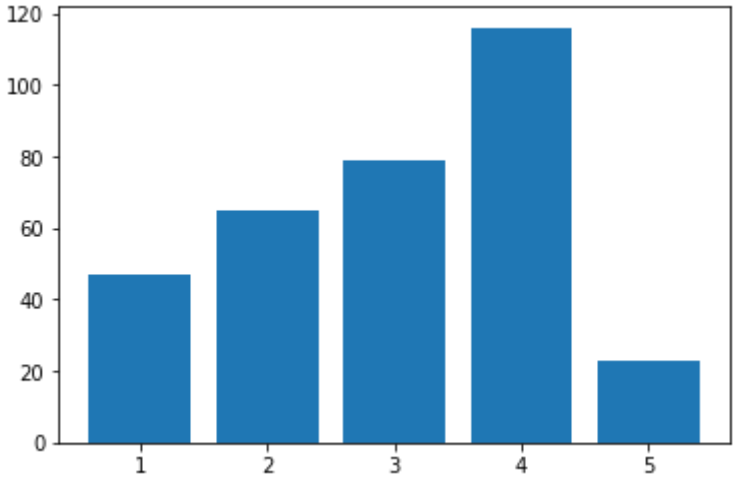
INSIGHT 1 - The y-axis is not a simple 0 or 1 excitation, we can't just put a 0-1 neuron at the end of the model, the values in y are dynamic taking anything in the range 0 to 2. We can also see that the first two tensors are positively ranged whereas the third and fourth are distributed around 0. ie (-0.5 to 0.5 and -0.75 to 0.75)

2 The count of excitations in each data point

Excitations in the X tensor



Excitations in the Y tensor



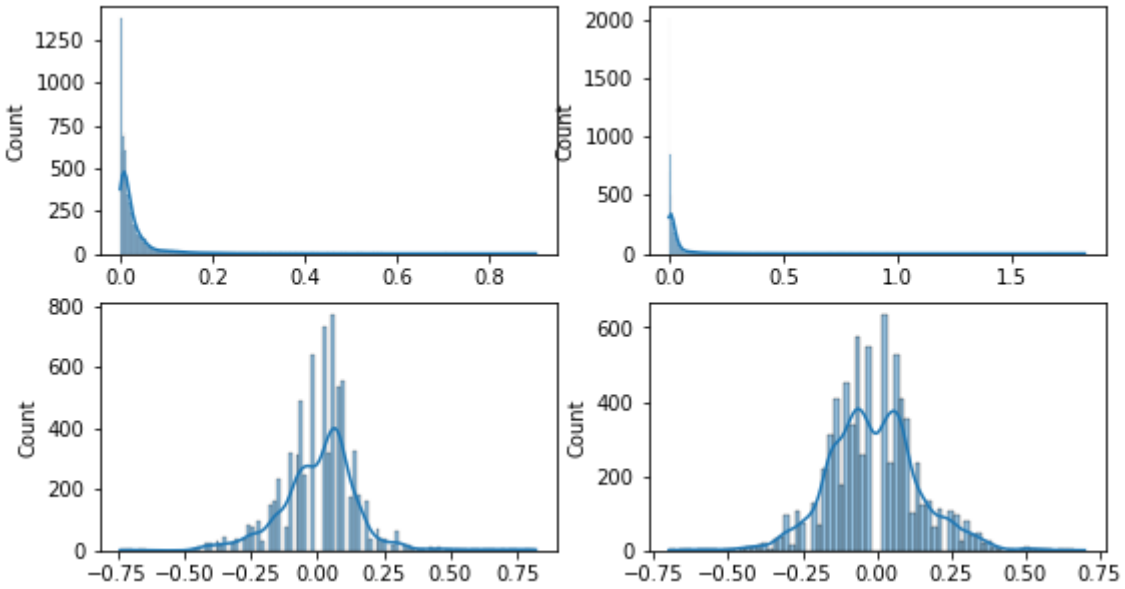
Continuing with the same 5 data points, I plotted the number of excitations in each point

INSIGHTS

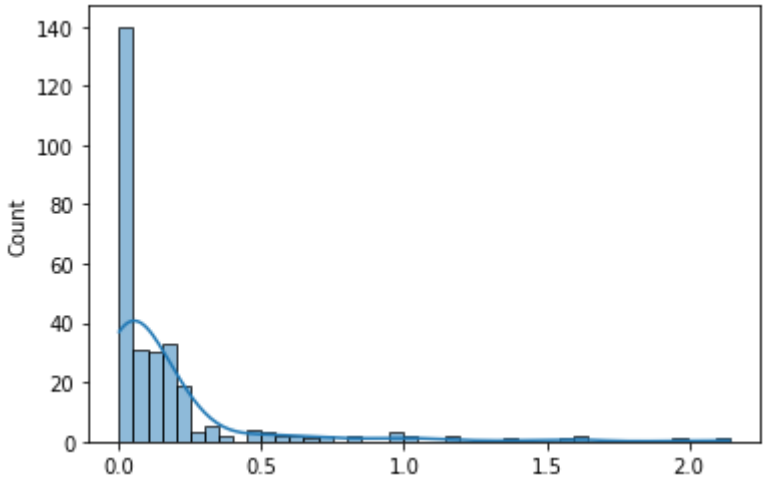
- Excitations are far more in x than in y
- Excitations in y depend on that of x (not always directly proportional)
- Curiously, even though there are up to 1500 excitations in x we only have up to 120 excitations in y
- We can't build our model, just as linear combinations we need some non-linearity

3. Histplot of non-zero values

In X



In Y



We can see that, in the first two tensors of X, the distribution is mostly after zero and in the last two, they are distributed across 0.5

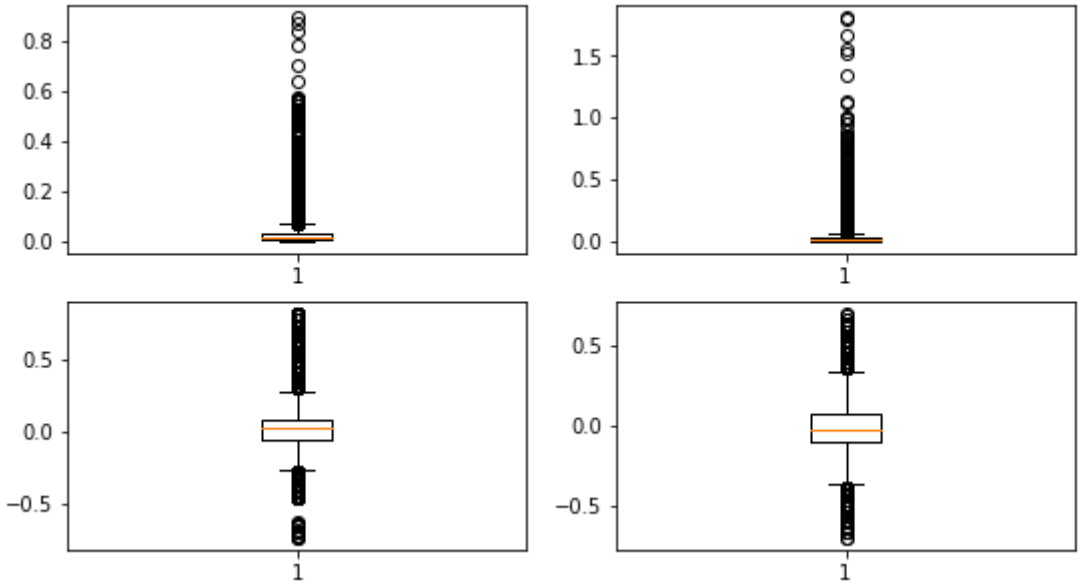
We kind of got this in the first graph itself but more importantly, here we understood that, the range of values and that the tail is very big in these distributions.

And also we can see that the values of the excitations are very few for one value, ie there are only a few 1.0s and 0.8s in the dataset.

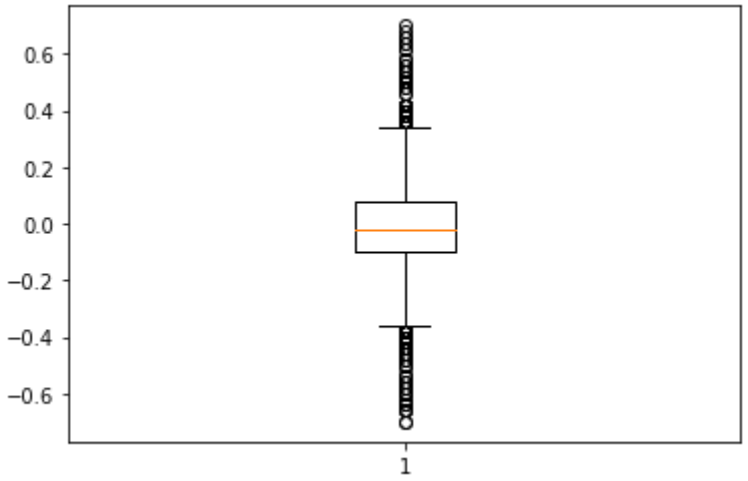
The model has to be very good at utilising these values and not just generalise it to all 0s

4. Box plot to re-affirm the above finding

X



Y

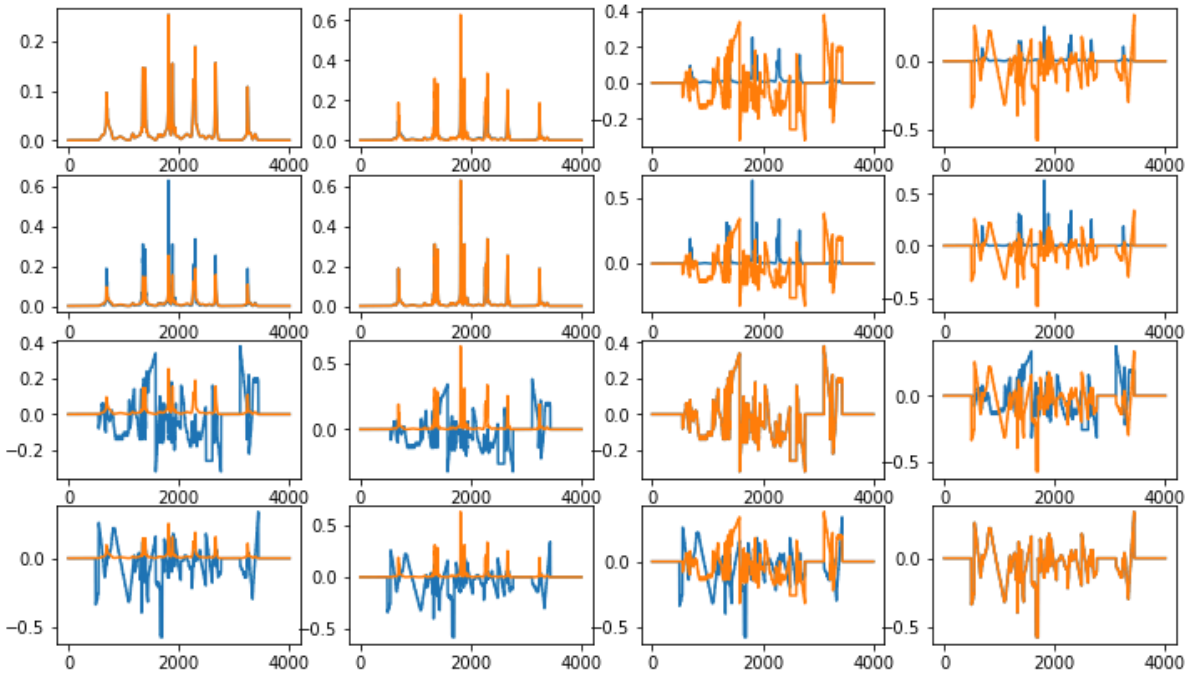


These box plots re-affirm that the non-zero values which are very close to 0 are as prevalent as 0s, therefore our model has to capture the excitations well and make use of it

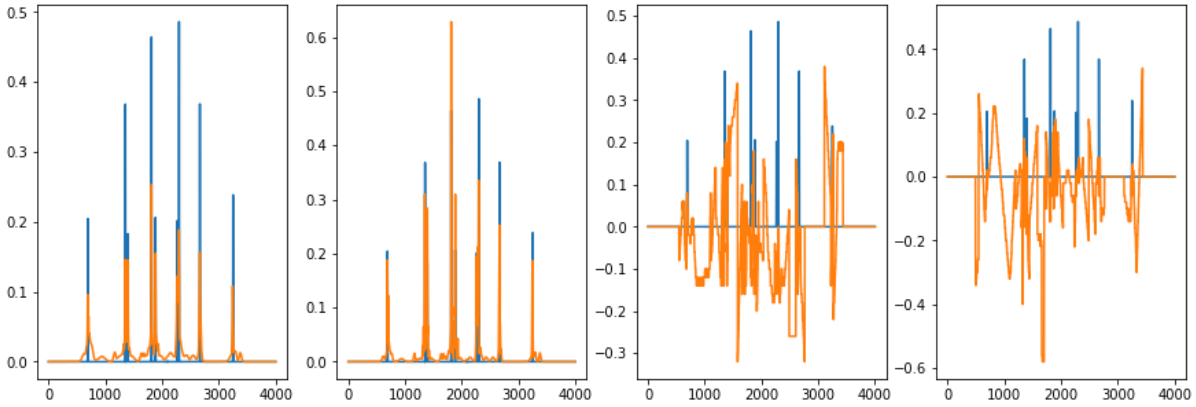
The excitations in the Y axis are also very sparse and have to be generated efficiently.

5. Correlation of features

Among X tensors



Between X and Y tensors



When we discussed the dataset in class we discussed that we don't know the relationship between the axes with each other or with the y tensor, so I plotted these graphs and the insights I got from these graphs are that -

1. The first two features in the x-tensor are in sync with each other
2. The second feature is a bigger excitation than the first one
3. The same is not true for the third and fourth rows of the X tensor they are independent
4. Also, there is no correlation between (1,2), 3 and 4

Note - This correlation is just in the random data I plotted, I'm not sure if this will hold true for the entire dataset

Also, another interesting insight is that we see that the y tensor has a good correlation with the first and second rows of the X tensor whereas the third and fourth rows in the tensor are just off.

With this understanding of the data, I started architecting my models.

NOTE - In the section below I answer questions 2, 3 and 4 together, ie, I show the three models I architected and the steps I took to counter overfitting or underfitting and how I tuned parameters as well as how the size of the sample size effects the performance of the models.

Questions 2, 3, 4 - My three models, hyperparameter tuning and impact of size

The split

Before we look at my models, I wanted to explain how I split the data

I split the data into the following sets

- Large Training - One with 60 samples for train, 20 for validation
- Medium Training - 40 training and 40 validation
- Small Training - 20 train and 60 validation
- Debugging Sample - 5 train and 5 validation (I tuned hyperparameters on this dataset)

I use the same split for all the three models

MODEL 1 - Fully Convoluted Autoencoder using Conv1D.

Novelty -

1. Using Conv1D instead of Conv2D as the x tensors are 1 dimensional.
2. Latent space is connected by convolutions, not a linear layer.

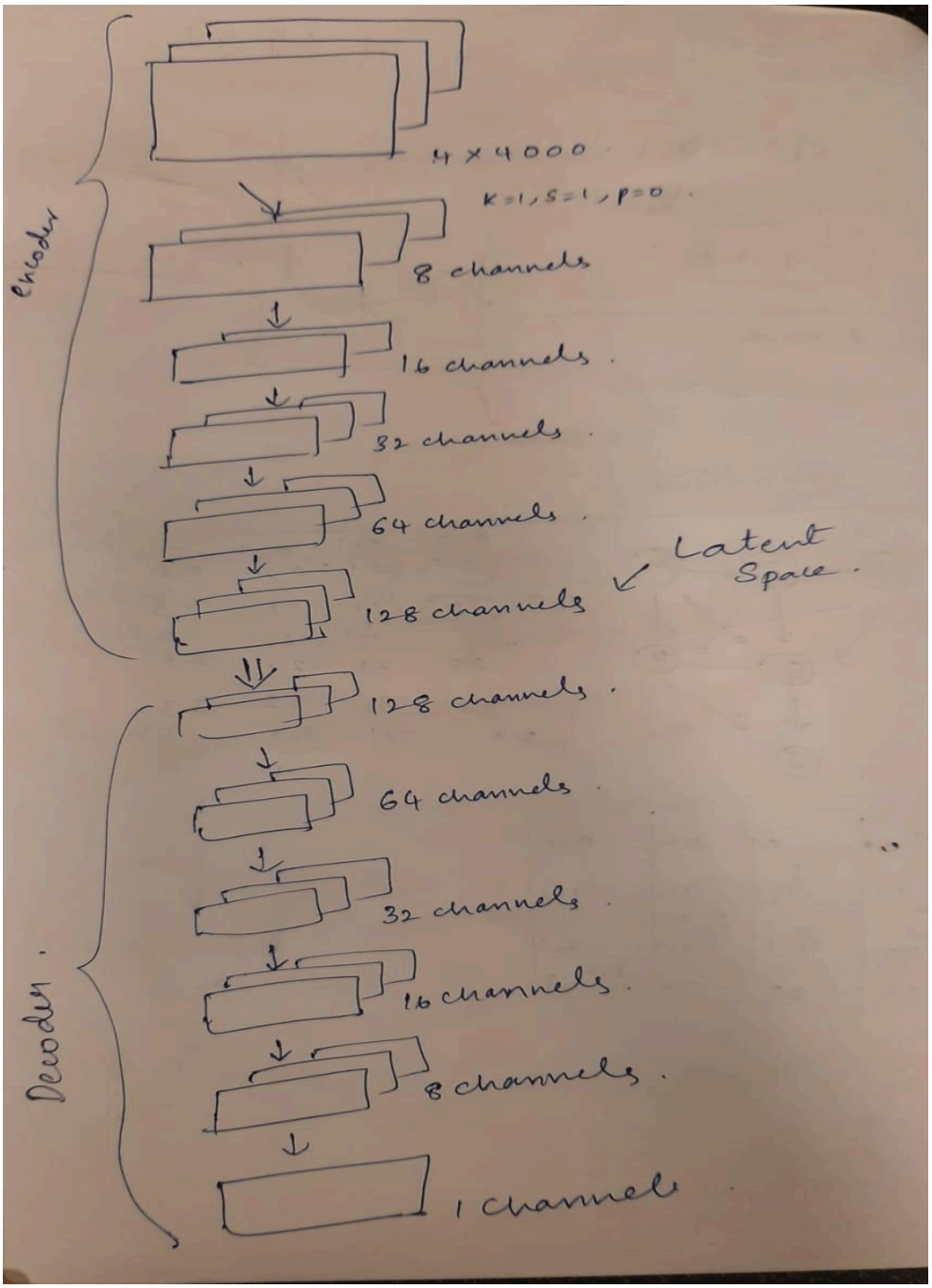
The architecture:

Just like we have 3 channels in an image, I considered each tensor in the X tensor as a separate channel and encoded into a latent space of 128 filters and decoded it from there.

As you can see in the architecture diagram below, the final output of the decoder is 1 channel.

A usual autoencoder would generally decode the data into the same dimensions but here I am doing a little differently as the output is in different dimensions (1 * 4000).

I actually tried to use a latent space using Linear layers but that was too heavy to train.



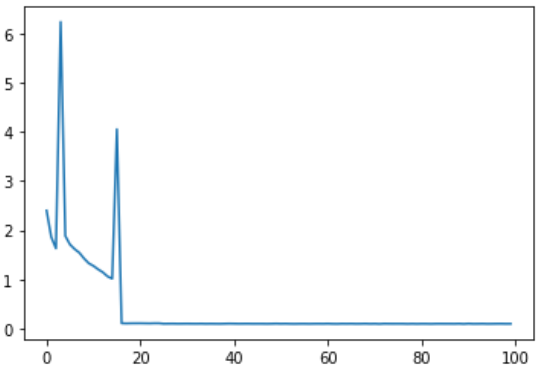
Let us now look at how I tuned the model using various hyperparameters.

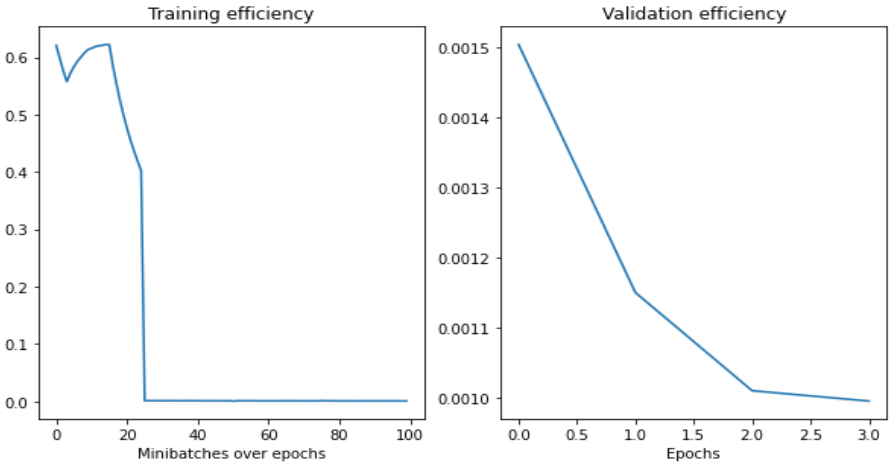
Tuning Hyperparameters -

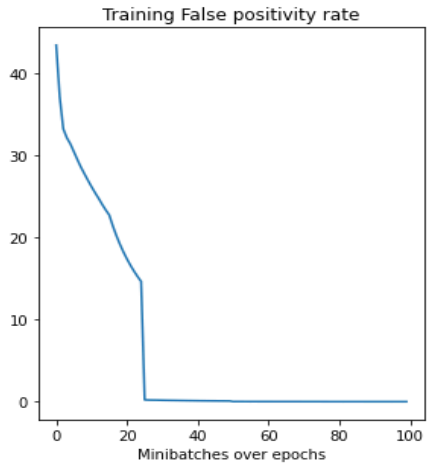
THE BASELINE

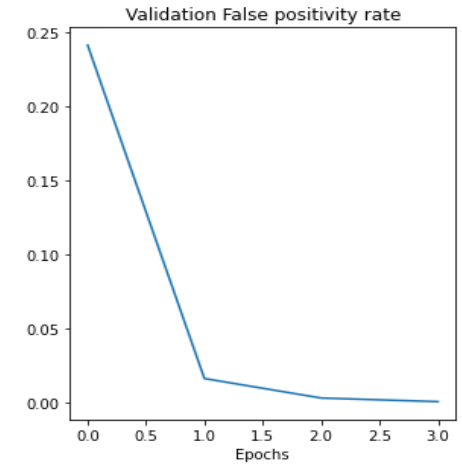
Num epochs = 4, Learning Rate - 0.001, Batch Size - 1000

Loss over training

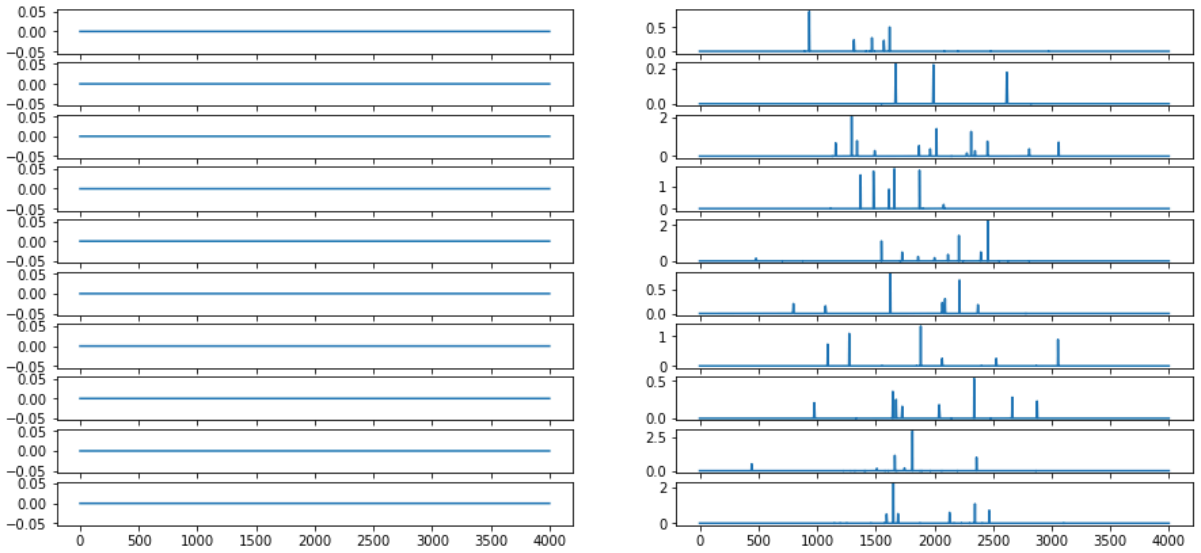








The Generated output

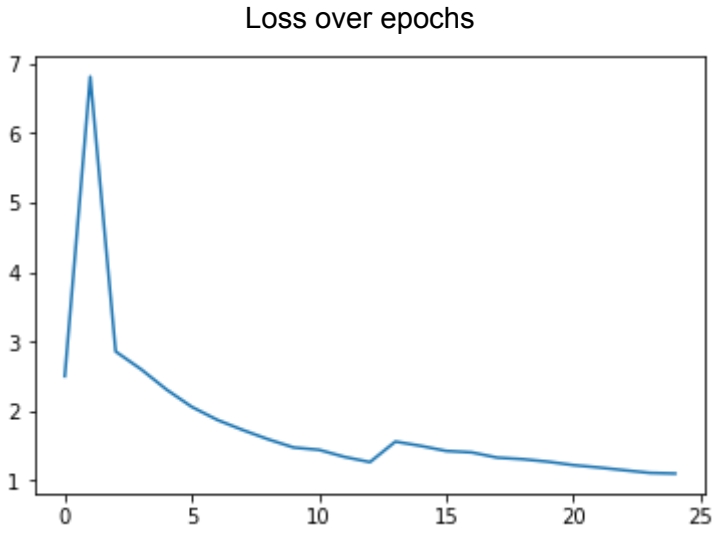


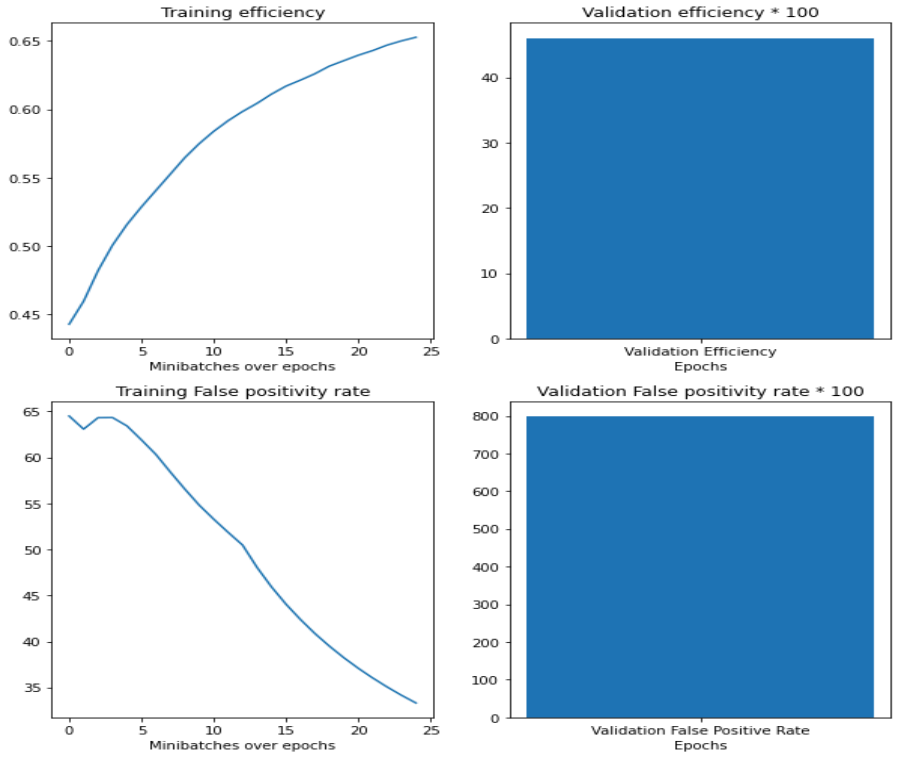
We can see from the above graphs that this setting has just created a model that just learns to output 0s.

ie, as we saw in the data visualisation section above, there are a lot of 0s in the y tensor and this model just learnt to output all 0s. A case of overfitting

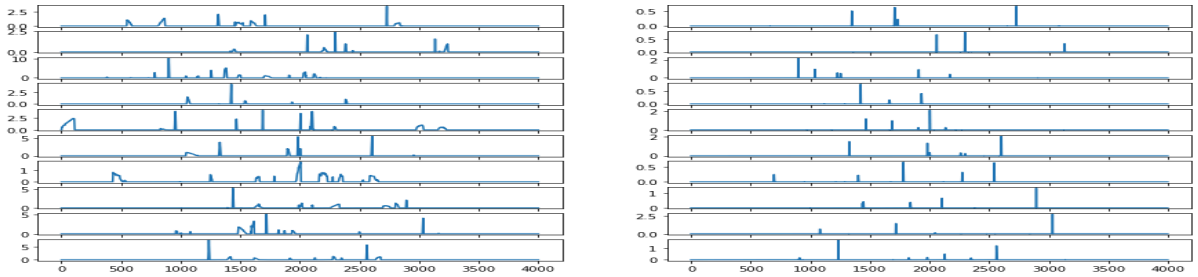
Tuning 1

To tackle this overfitting problem, I reduced the number of epochs to 1





Generated output



Now we can see that, the model is not just predicting 0s and generating some kind of output, also the loss is going down, efficiency is increasing and false is reducing as well.

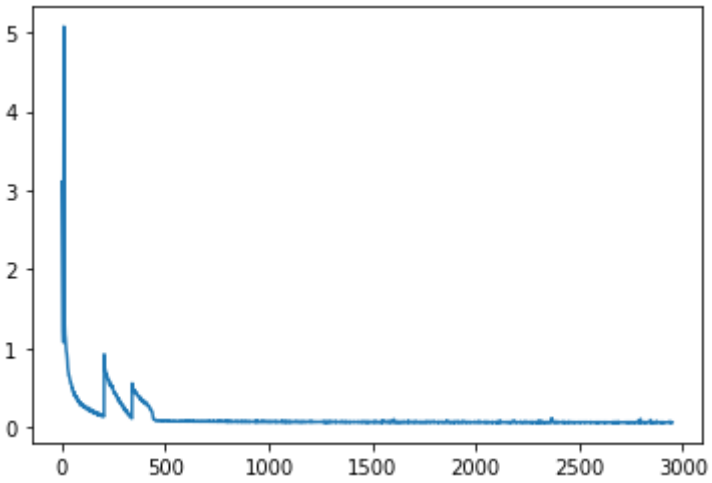
Tuning 2 -

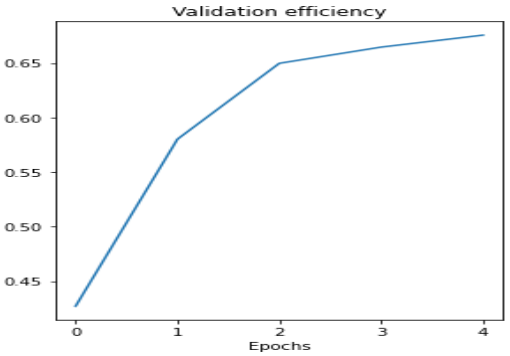
Earlier I didn't have a 128-neuron layer as the latent space, I had a 64-neuron layer, now I added another layer and made it the latent space.

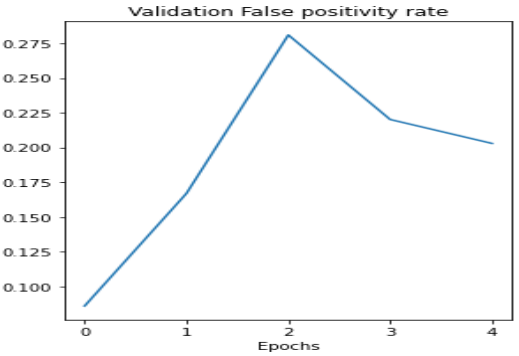
Here I also trained it on the **large dataset** ie 60 training and 40 validation

I also reduced the batch size as well to 500

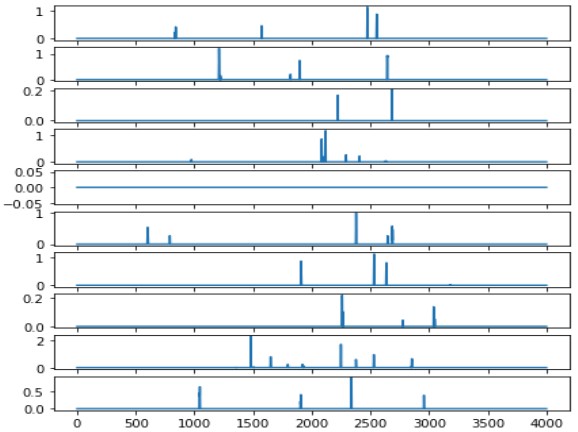
Loss over epochs

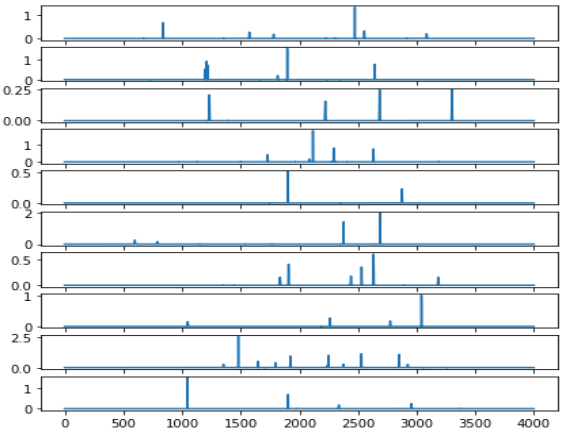






THE GENERATED OUTPUT





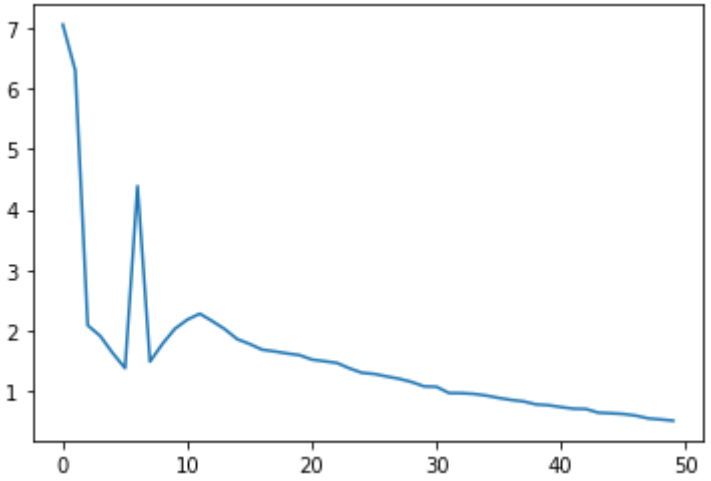
Now we can see that the loss has become more or less 0 and the output of the model as well is very good the range of the output is also close to what the original data was supposed to be

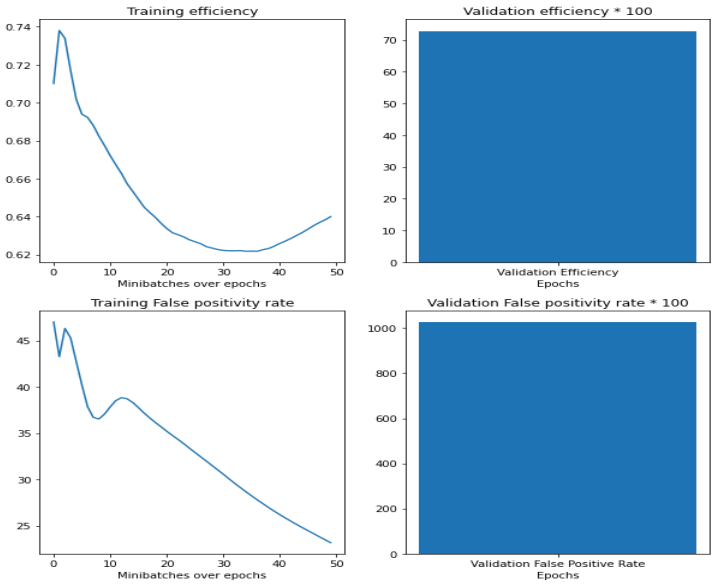
Seeing that efficiency was hit badly after 2 epochs and fp rate increased after 2 epochs, I reverted back to 2 epochs.

Trying different optimisers

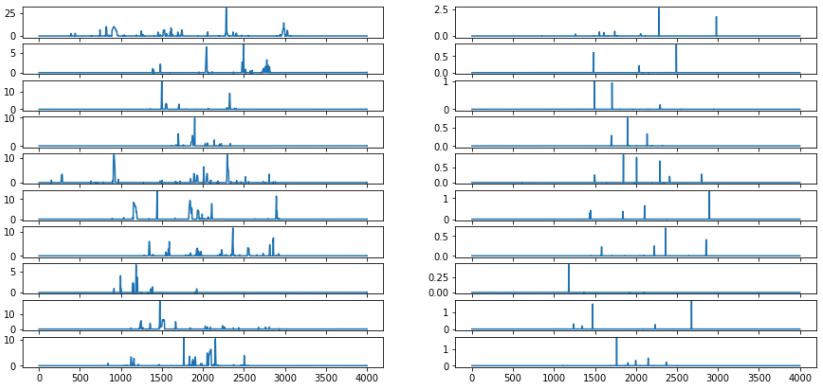
ADAM

Loss



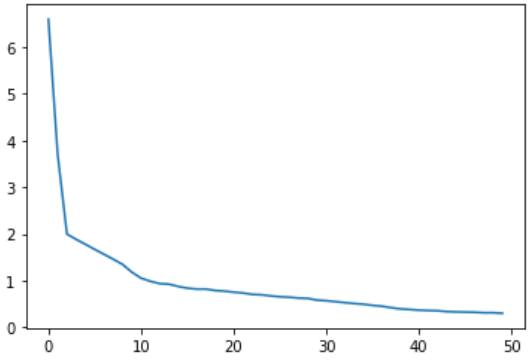


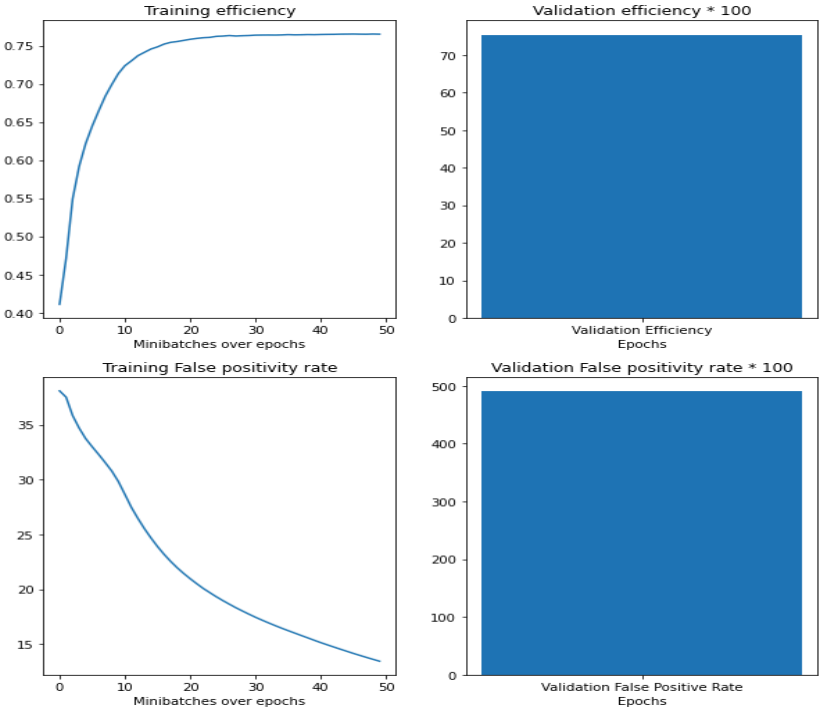
Generated Output



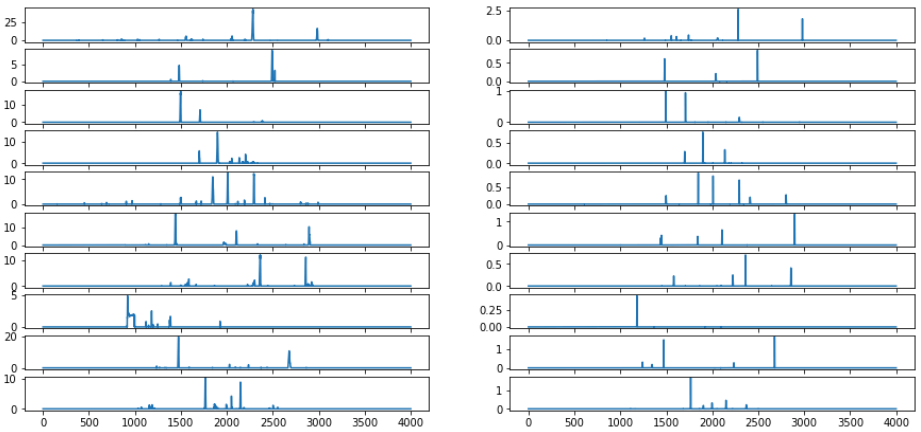
We can see that adam converged faster than the basic SGD Optimiser

RMS Prop optimiser





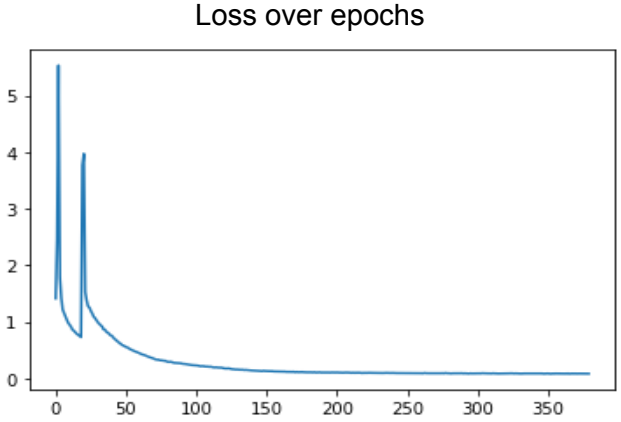
GENERATED OUTPUT

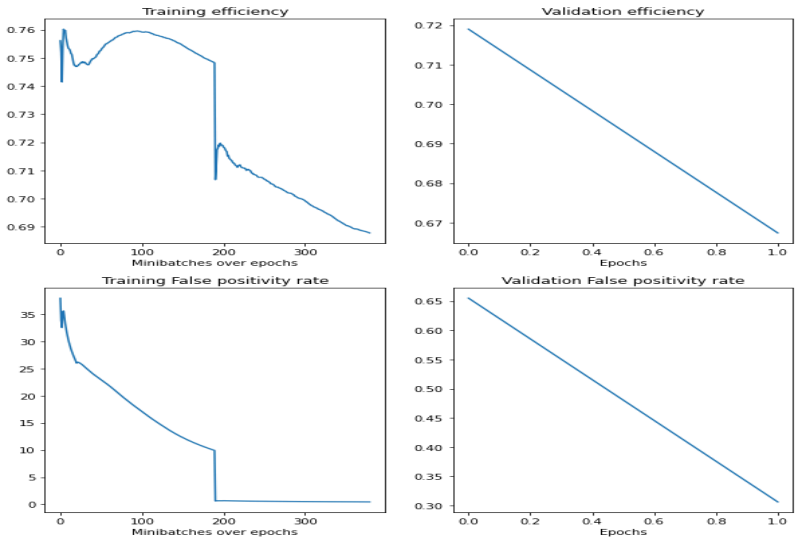


RMS Prop outperformed both the Adam and SGD Optimisers

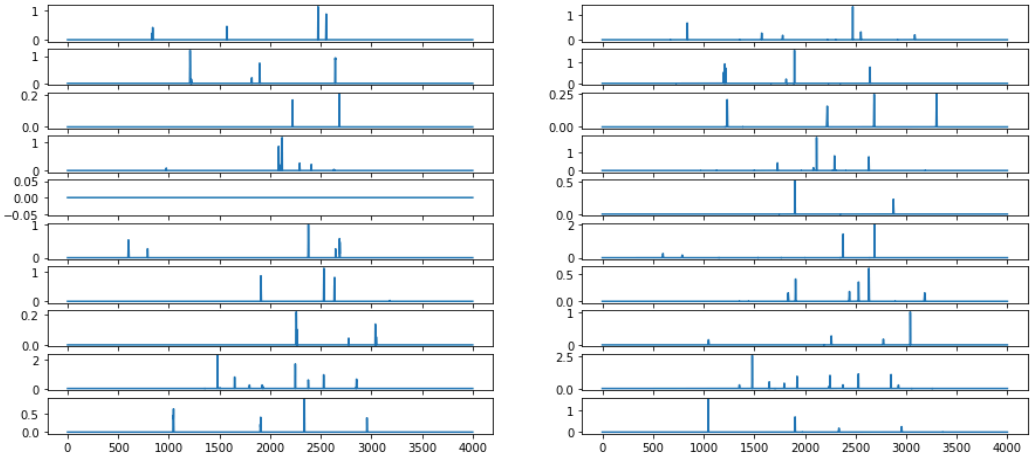
Since I trained the first model on a large dataset with SGD, I reverted back to SGD to compare the performance on different sizes of datasets and these are my findings

SMALL DATASET (20 training, 40 validation)



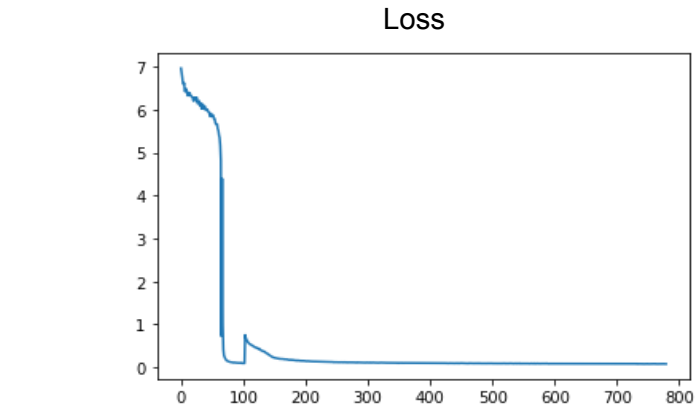


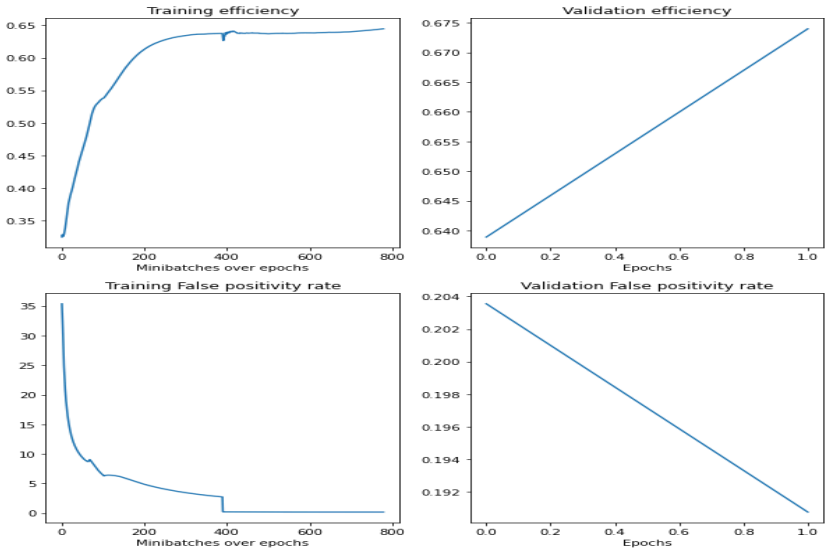
THE GENERATED OUTPUT



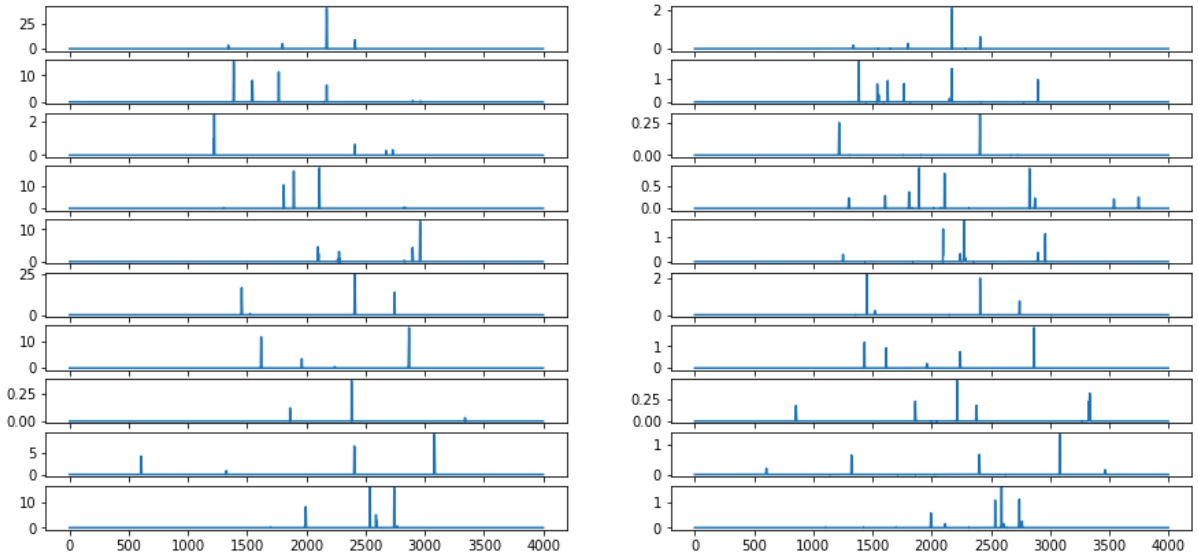
Though the generated output looks good, on closer observation, it is not as accurate also, efficiency and the false positivity rate took a hit because of the small size.

MODERATE DATASET (40 train, 40 validation)



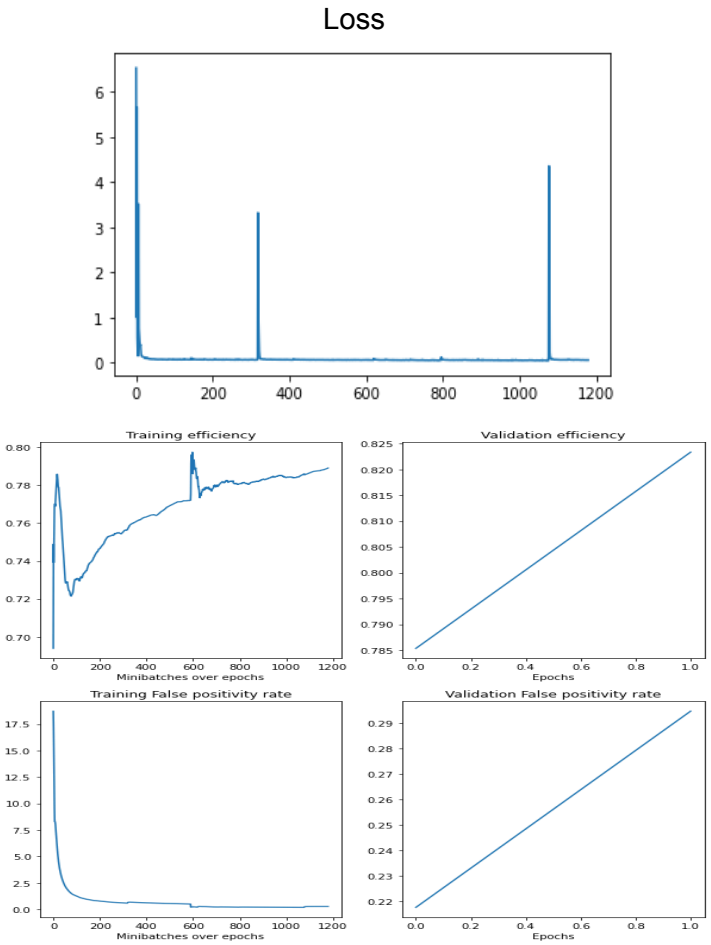


Generated Output

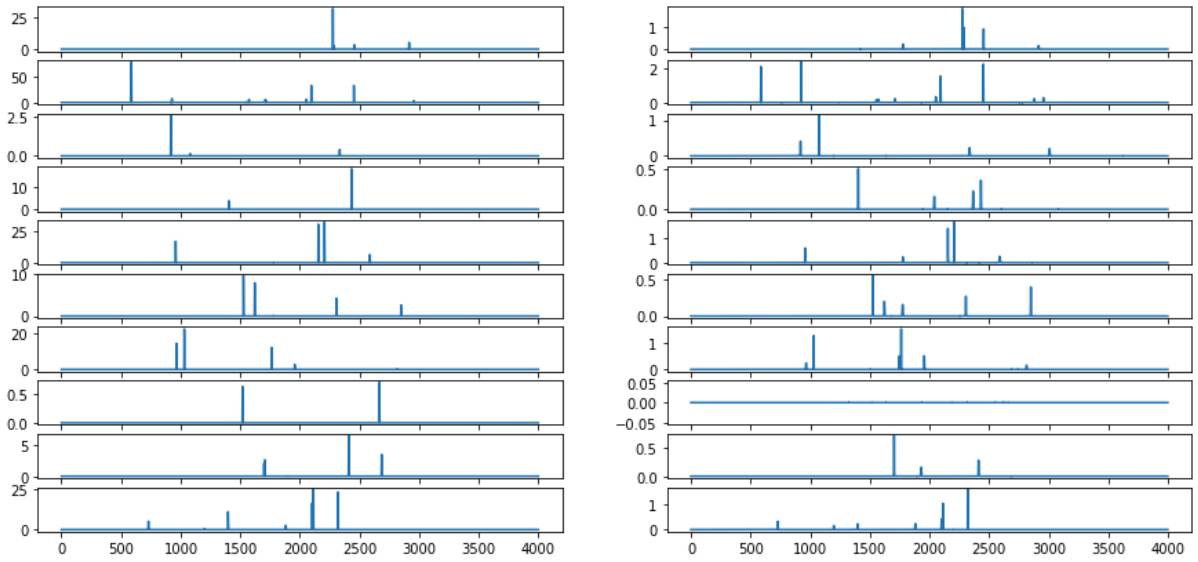


Even for a moderately large dataset, it was able to generate good efficiency and false positivity rate at a good loss as well, this was a good model as well

Since I got a very good performance on RMS Prop I trained my best model with RMS Prop for 2 epochs on a large dataset and the results are as follows



Generated Output



This model with RMS prop outperformed, the model with SGD Optimiser, It generated close to 82% efficiency and the false positivity rate was also low at 28% and also the loss is very small at 0.06. These were the best hyperparameters for this model that I was able to tune to.

MODEL 2 - The Encoded LSTM

Since Conv1D is used for text and LSTM is used for text as well, I tried LSTM RNN for the second model.

Novelty ->
But since we were looking for novelty and encoding worked well for me, I tried to use it in this architecture as well. Also I though passing the complete 16000 items to LSTM would be computationally very heavy (I tried it and it was heavy, I report it below)

Instead of having an embedding layer, I encoded the X features into a latent space, passed this into the lstm, and then decoded the output of the lstm.

Also, the lstm is itself one kind of generator (ie Seq2Seq model), since the input is not equal to the output dimensions the final cell of the RNN would generate the required output.

Trial 1

I tried passing the whole 16000 samples directly into the LSTM and since the lstm has a long memory to hold in its cells it failed,

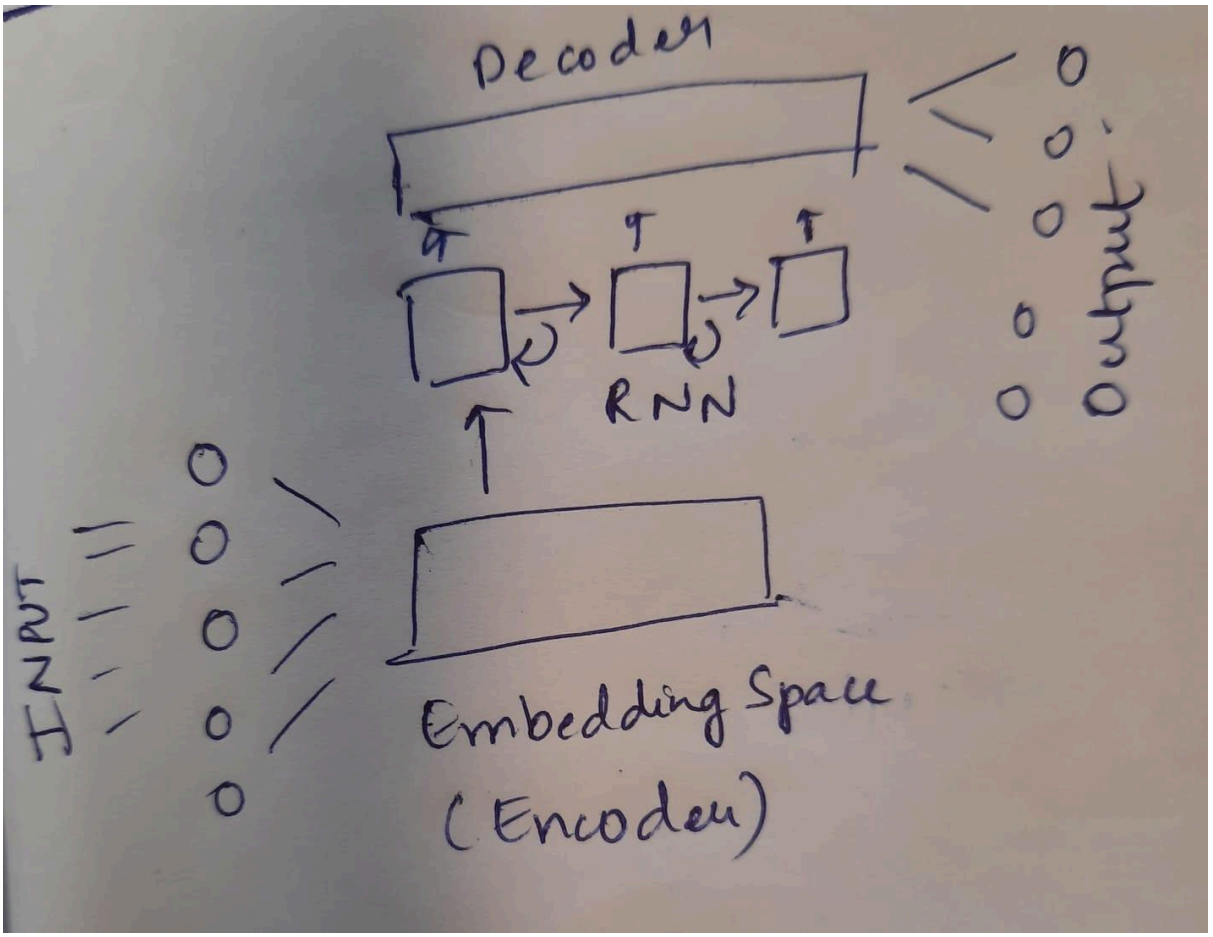
Colab's GPU was not able to keep up as there was no embedding dimension and it kept on crashing.

Trial 2

Having achieved decent success with encoder-decoder architecture in the previous architecture, I tried using the same as the embedding layer for the LSTM.
I was not really sure if it would converge as the traditional embedding layer is used to embed text as vectors and not really create a latent space.

With this doubt in mind, I anyway went ahead with this architecture let's look at the results.

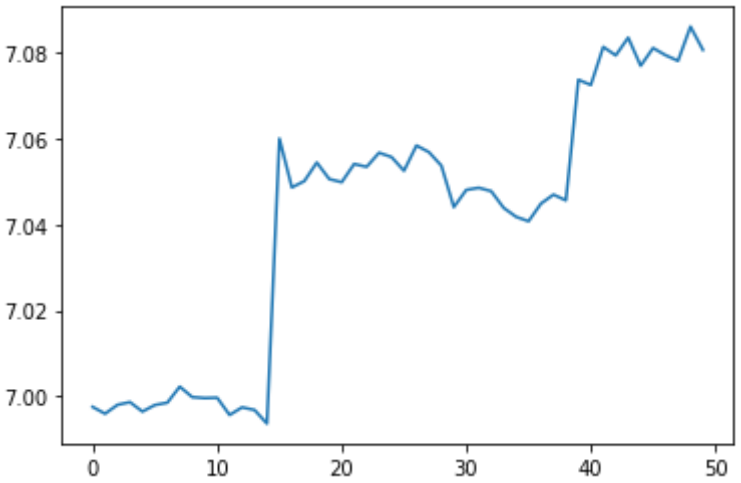
THE ARCHITECTURE

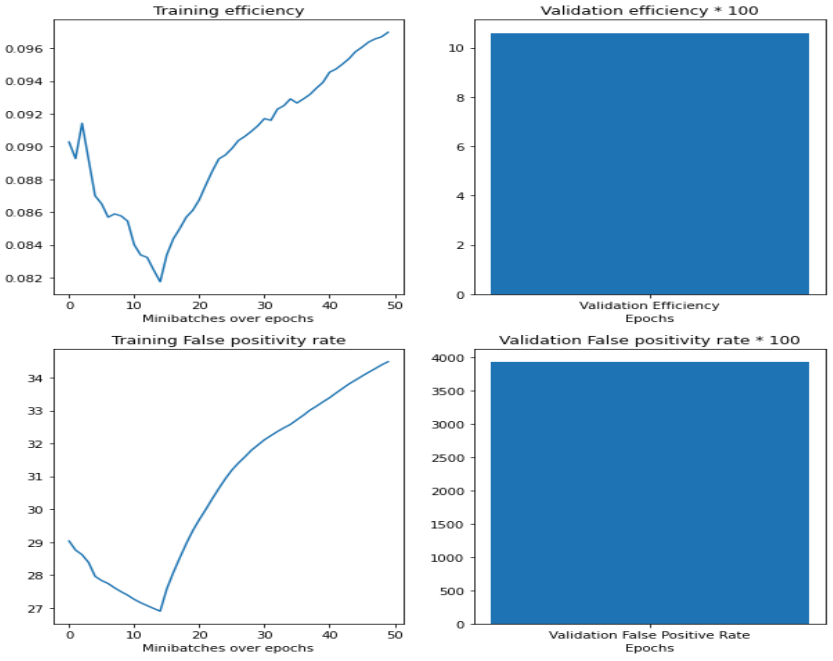


Trail 1

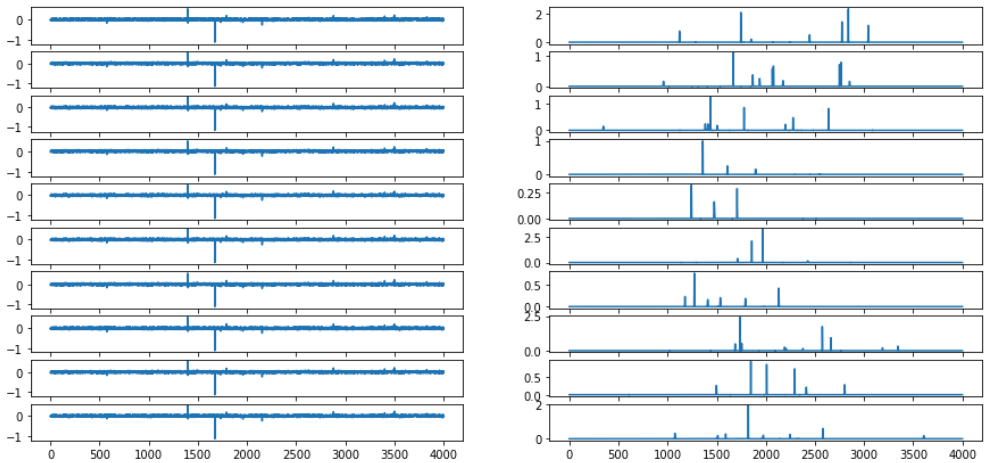
Epochs - 1, LR - 0.001, Batch Size - 500

Loss





Generated Output



As we can see, the loss just kept on increasing, and efficiency and false positive rate also kept on increasing,

When we look at the output, the generated output is **SAME** for all the inputs

This could be attributed to the memory that each cell in the LSTM holds.

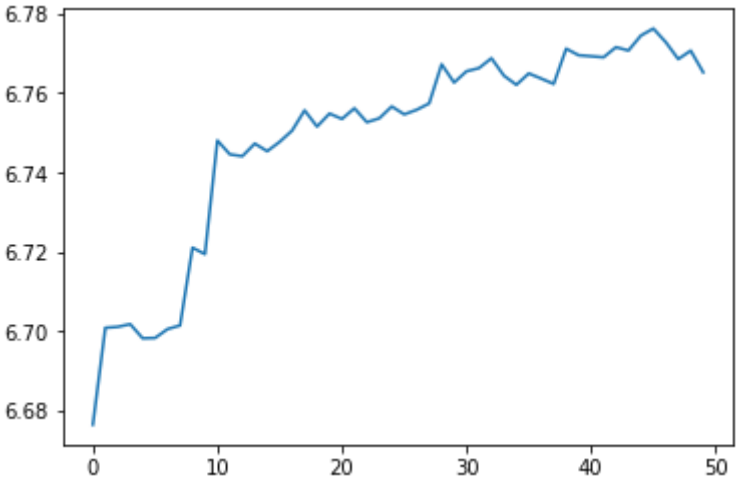
Because of the stored memory, the decoded output and the final output were all same.

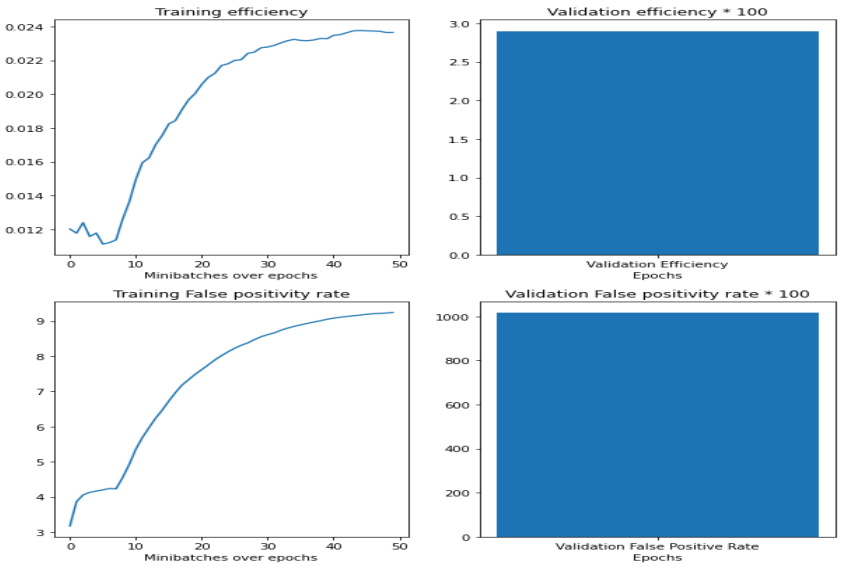
HYPER PARAMETER TUNING - 2

Tweaking the architecture -> Increasing the embedding dimensions

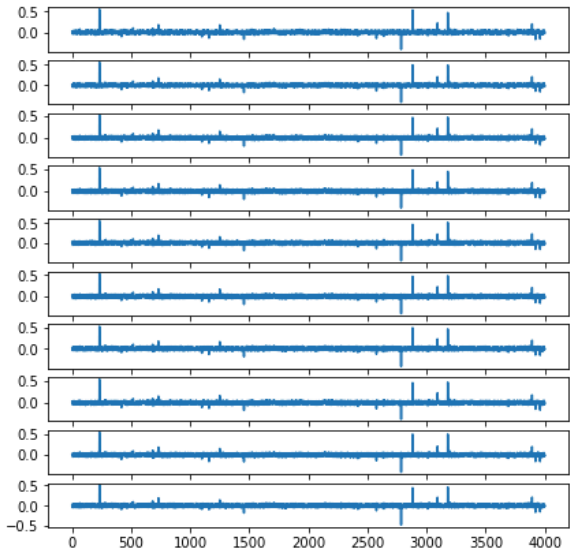
I thought the issue was with the architecture and tried adding more complexity to it.

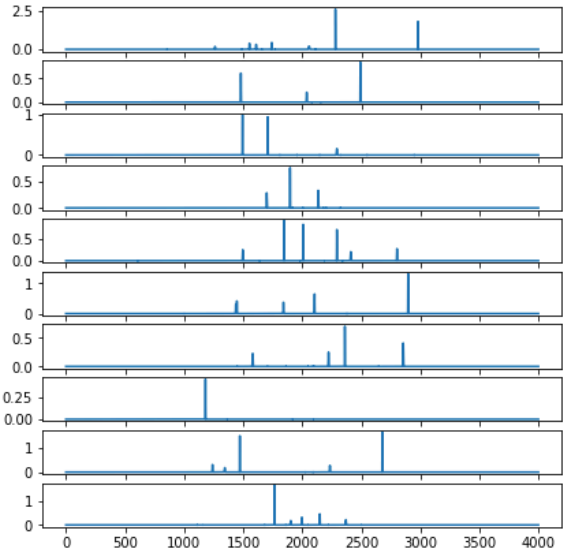
Loss





Generated Output

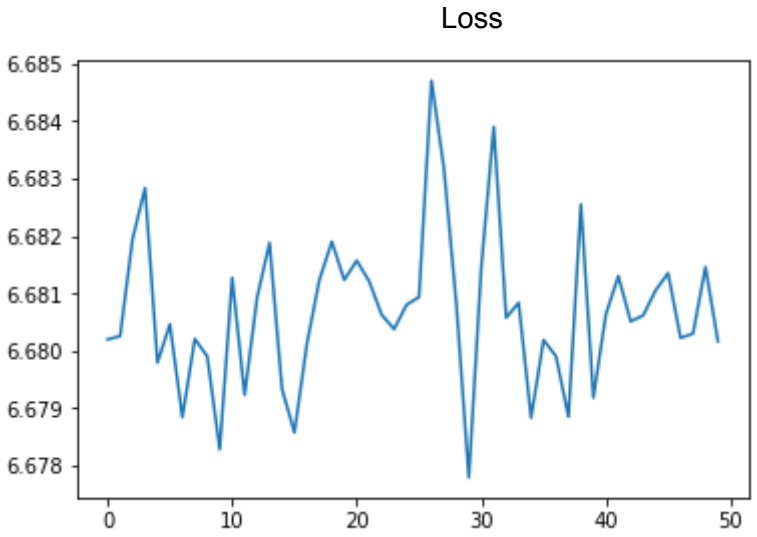


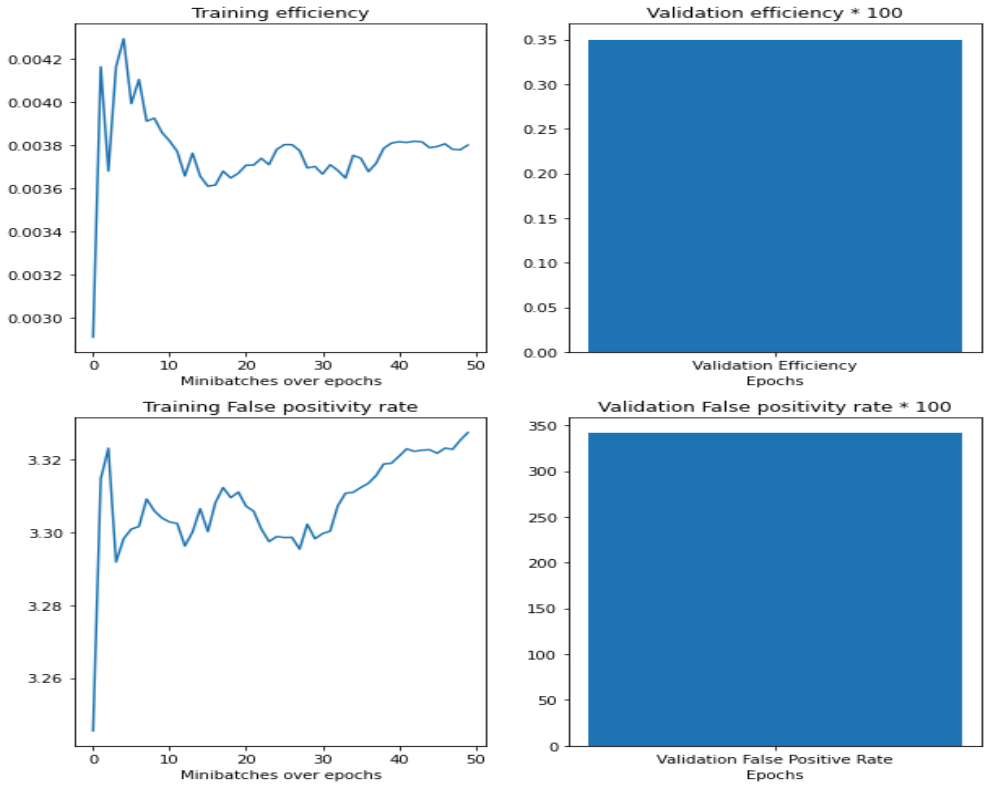


This tweak was also a failure as the outputs were getting generated from memory and the density of the events also increased.

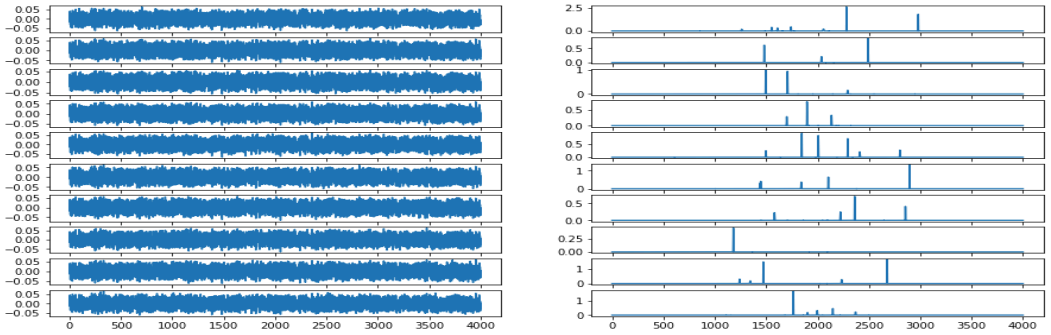
HYPER PARAMETER TUNING - 3

Reducing the learning rate to 1e-05





Generated Output

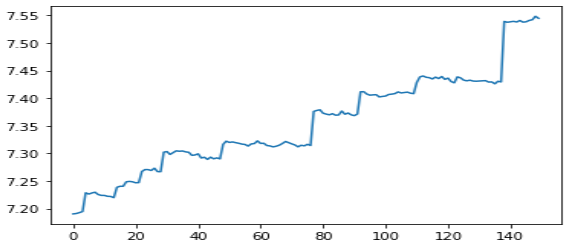


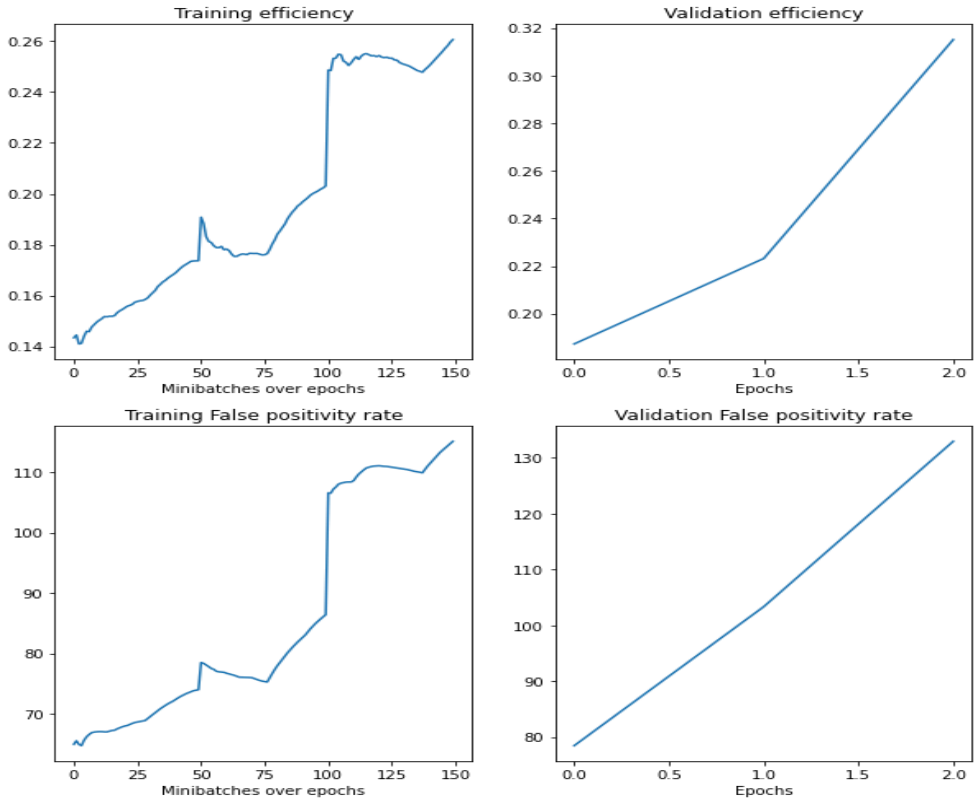
Here we can see that reducing the learning rate made the model excited for all the inputs, it didn't learn anything at all.

HYPER PARAMETER TUNING - 4

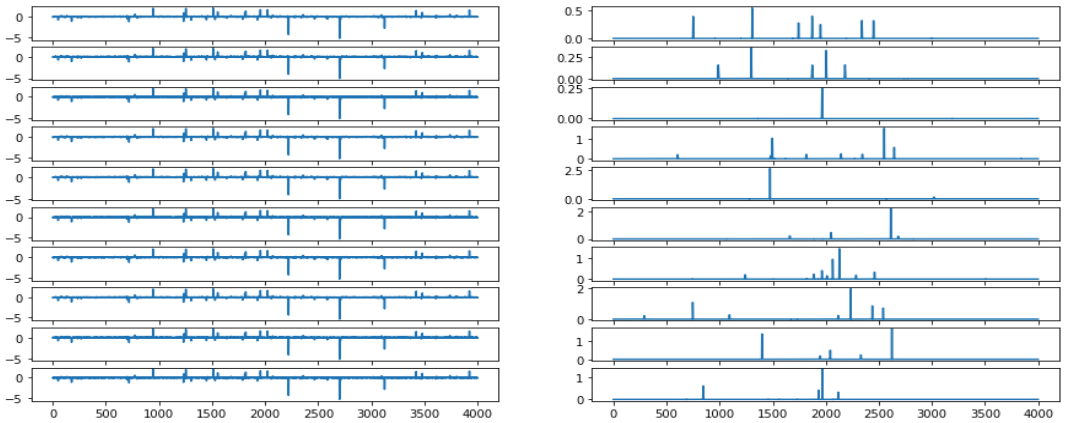
Falling back on learning rate and increasing epoch

Loss





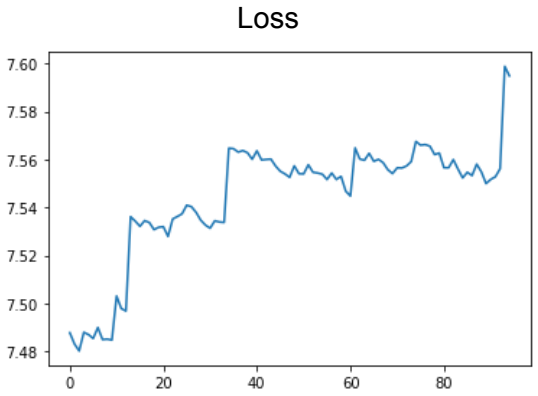
Generated output

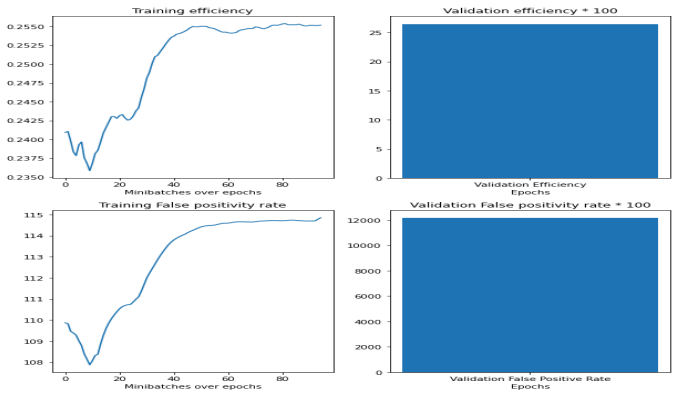


With this change, we reached our previous stage ie the excitations were coming out of memory and were the same for any given input

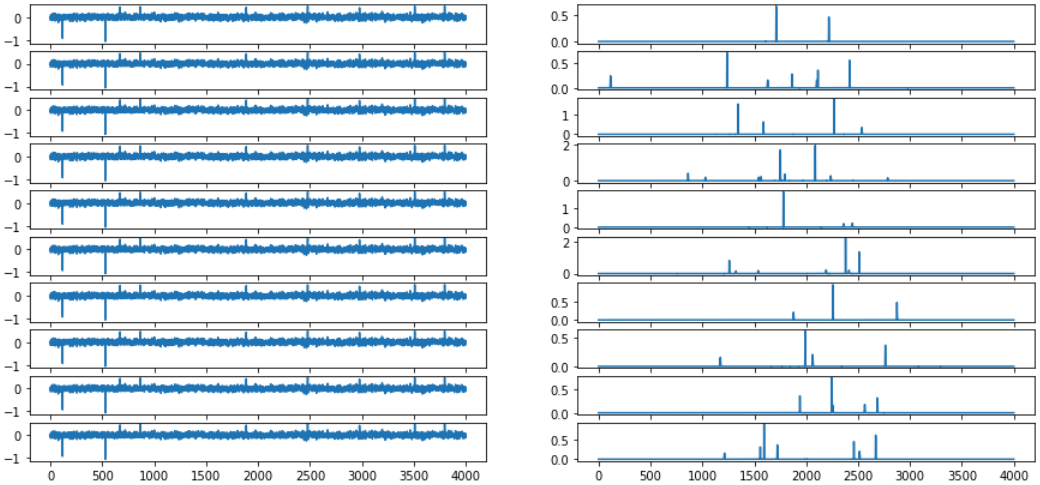
With these hyperparameter tuning, I thought I would see some improvements when I trained on more samples of data and I tried my three sets of data on this architecture.

Effect of training on small input:



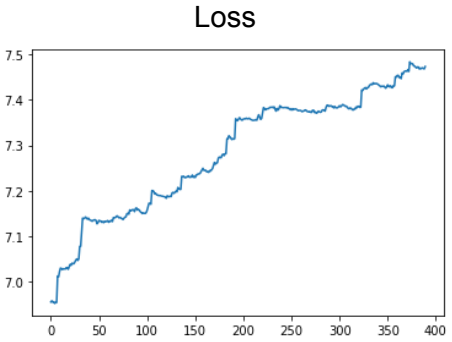


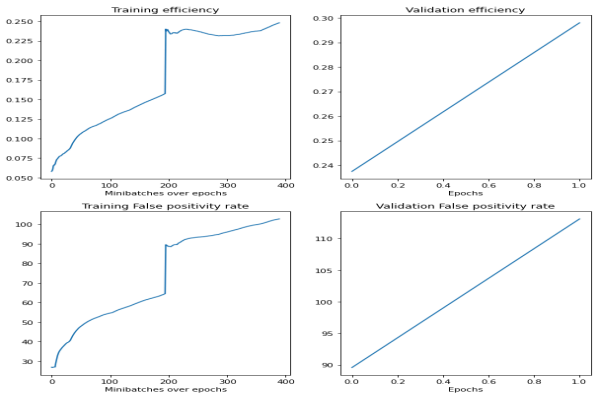
Generated output



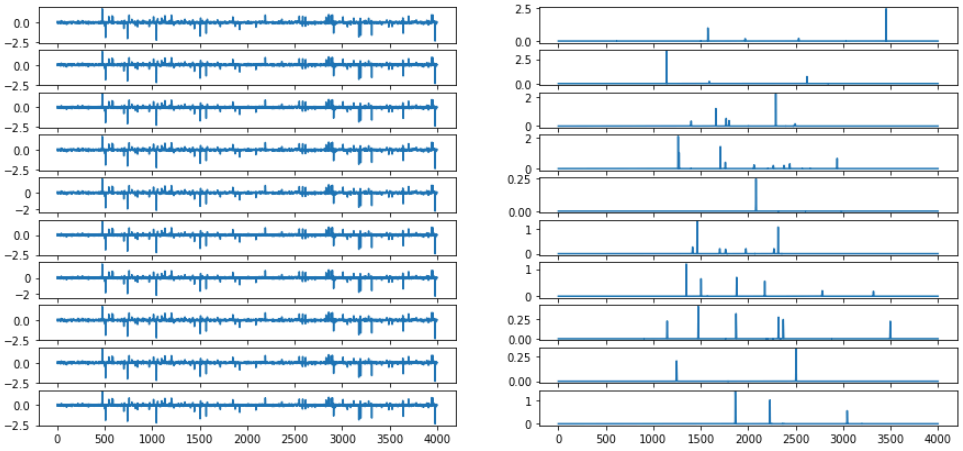
We can see that even though we put all the parameters the same, training the same input on the small dataset increased the overall excitations whereas the model was outputting from its memory that too same output for every input.

Impact of the medium dataset





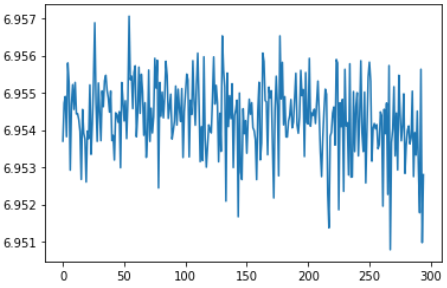
Generated output

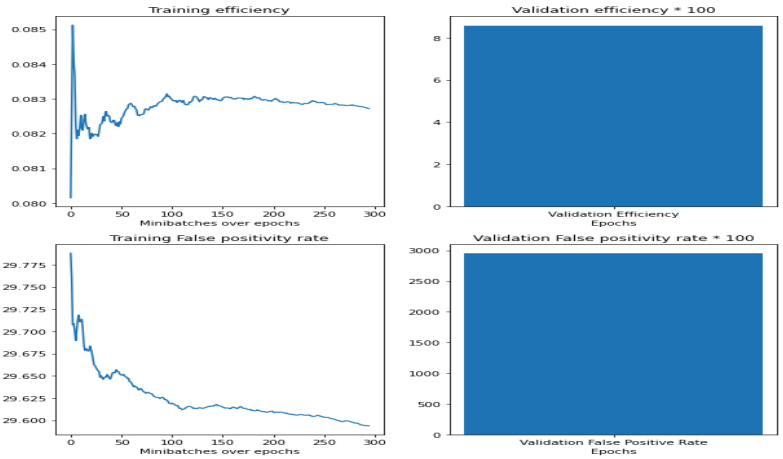


Though the loss was a little better than the smaller dataset but it was still outputting it out of memory which was bad

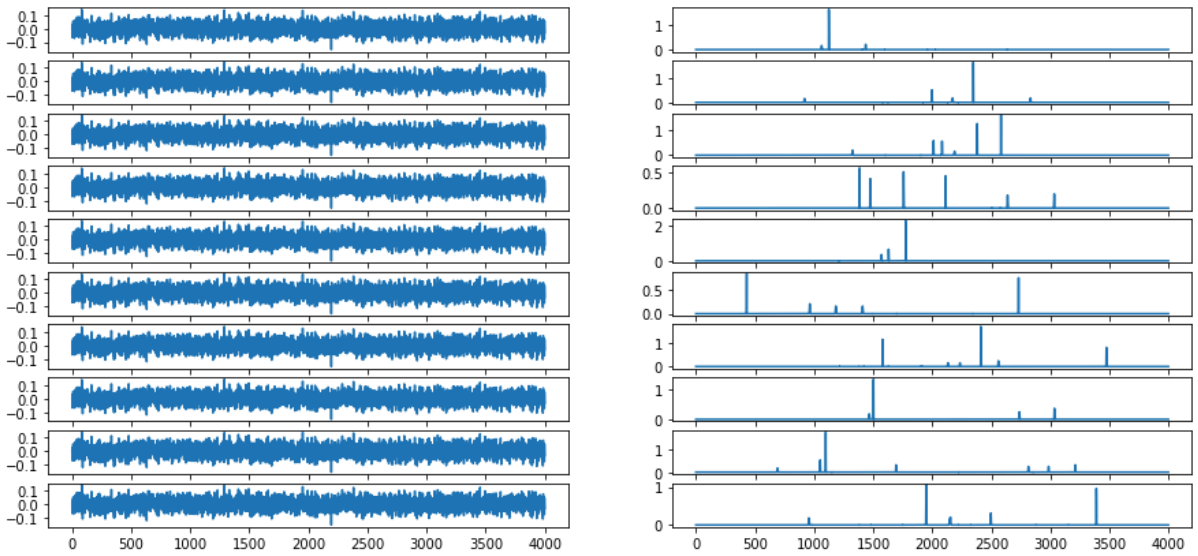
Impact of a larger dataset.

Loss





Generated Output



Even though the efficiency and false positive rate look like they are going in the proper directions, when we look at the output, the model just resorted to getting excited on all the inputs.

I think this is because the encoding of data points in the first layer was causing issues, I think this architecture (Encoder + LSTM) didn't really work well for this dataset.

MODEL 3 - ResNet with a Reducer

Since memory didn't work well for me I tried an architecture where it could forget the things that it learned.

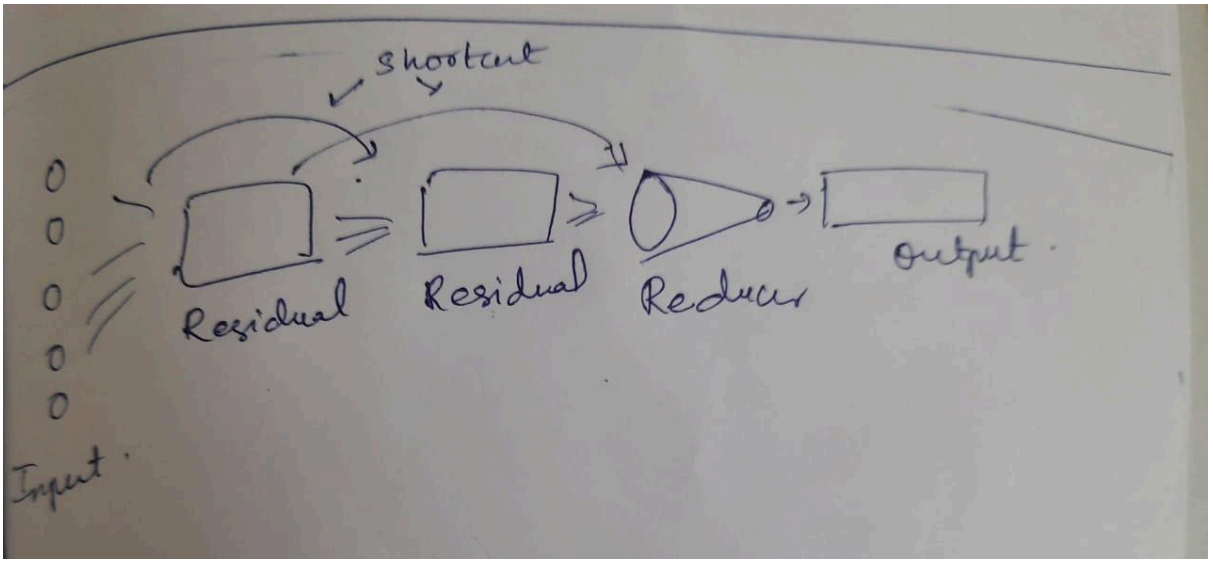
One such architecture that we learned in this course was the ResNet architecture also, in my first model, Conv1D worked pretty well for this dataset so I replaced the Conv2d with Conv1d

Novelty - In this architecture at the end of the residual blocks I pass the x + shortcut values to a standalone conv1d layer which takes the 4-channel output and converts it into the required 1-channel output. I called this layer **"THE REDUCER LAYER"**

The rationale behind using the reducer -

The output of the second residual layer in my adaptation of ResNet was just 1 channel just like we wanted however, the shortcut from the previous residual is 4 channels, so the issue when we combine both of them is it would become 4 channels again, so to get it back to 1 channel and make it same as expected output I used 1 channel output Conv1D and since it was reducing the number of channels, I named it Reducer Layer.

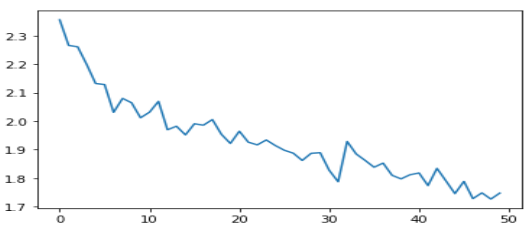
Also, the encoder with conv1d worked very well for me in the first model so I thought Conv1D would be a better reducer than just a normal linear layer.

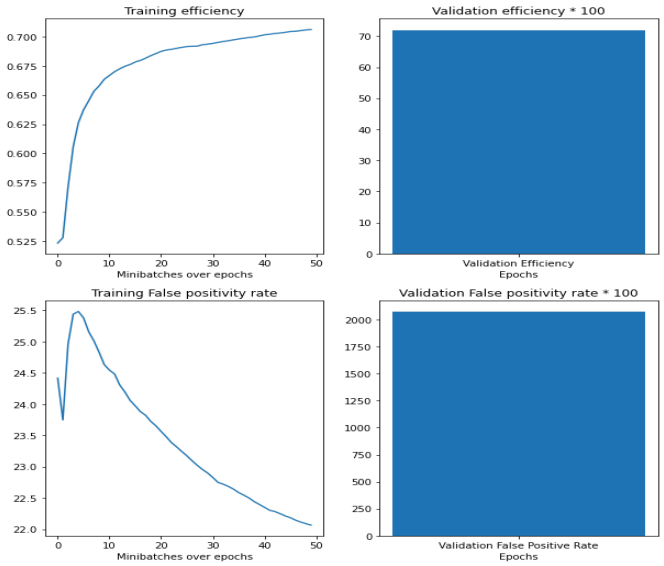


Trial 1 -

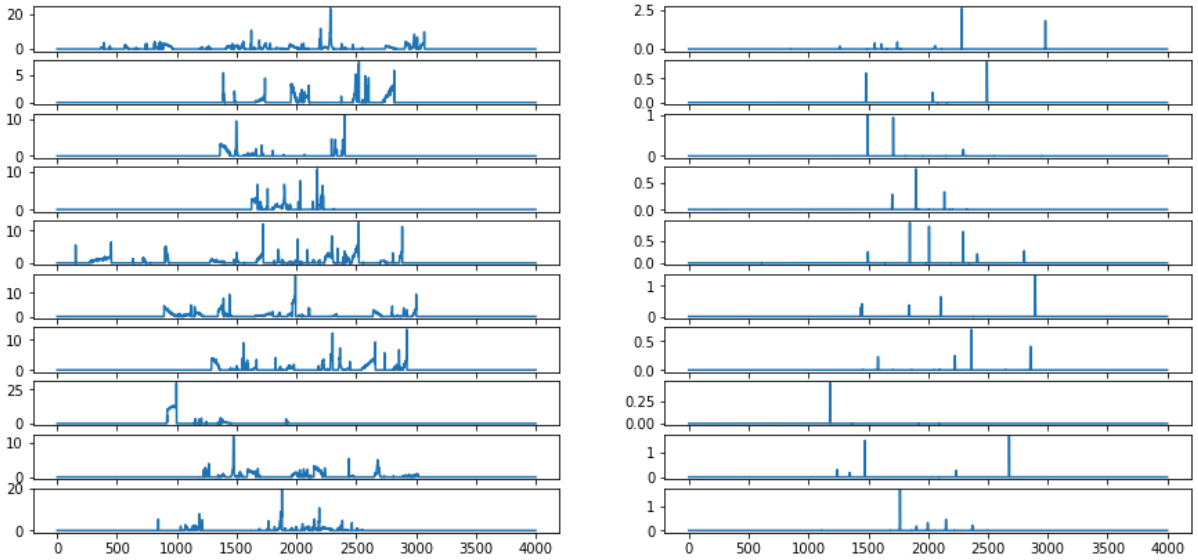
Epochs - 1, Learning Rate - 0.001, Batch Size = 500

Loss



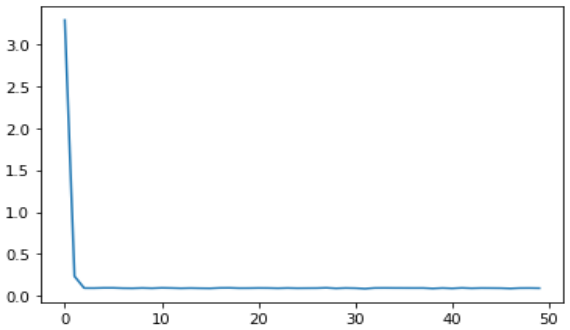


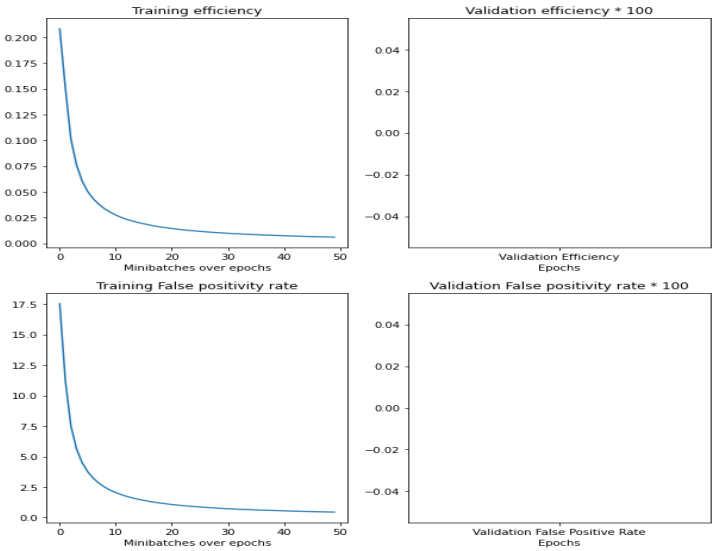
Generated Output



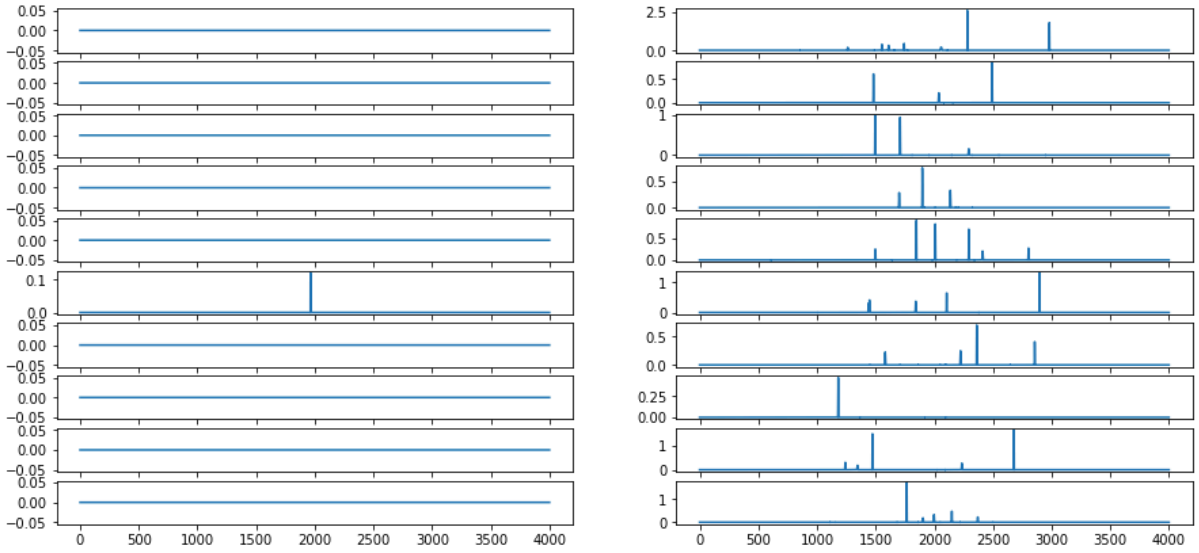
The loss was going down for this architecture, the efficiency was going up as well as the false positive rate was going down, everything looked fine but when I plotted the generated output, I learned that there were way more excitations than were required.

HYPER PARAMETER TUNING - 1 Increase Learning Rate to 0.1



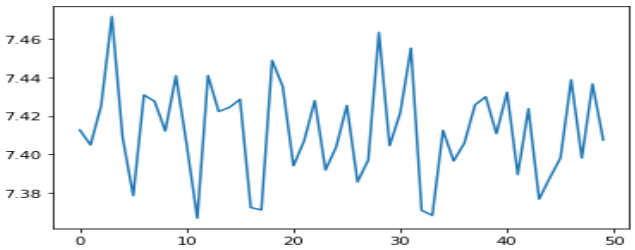


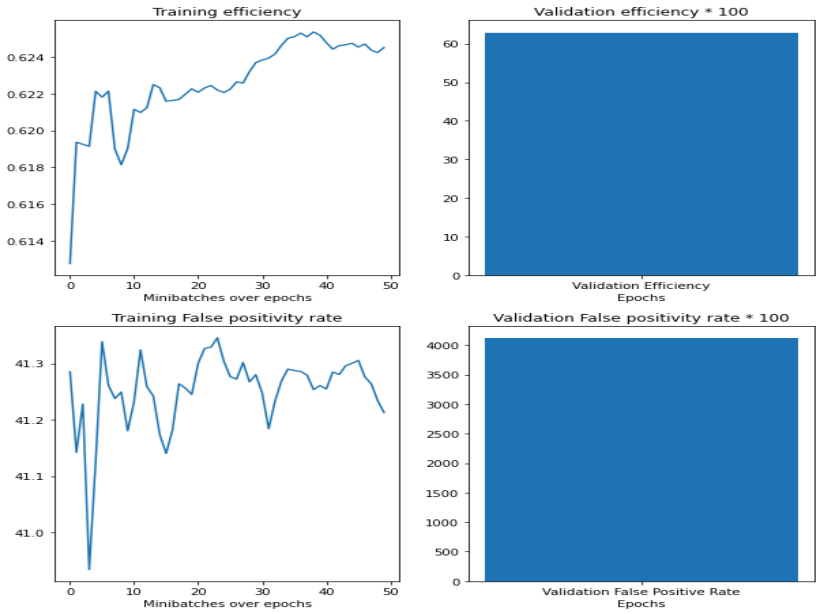
Generated Output

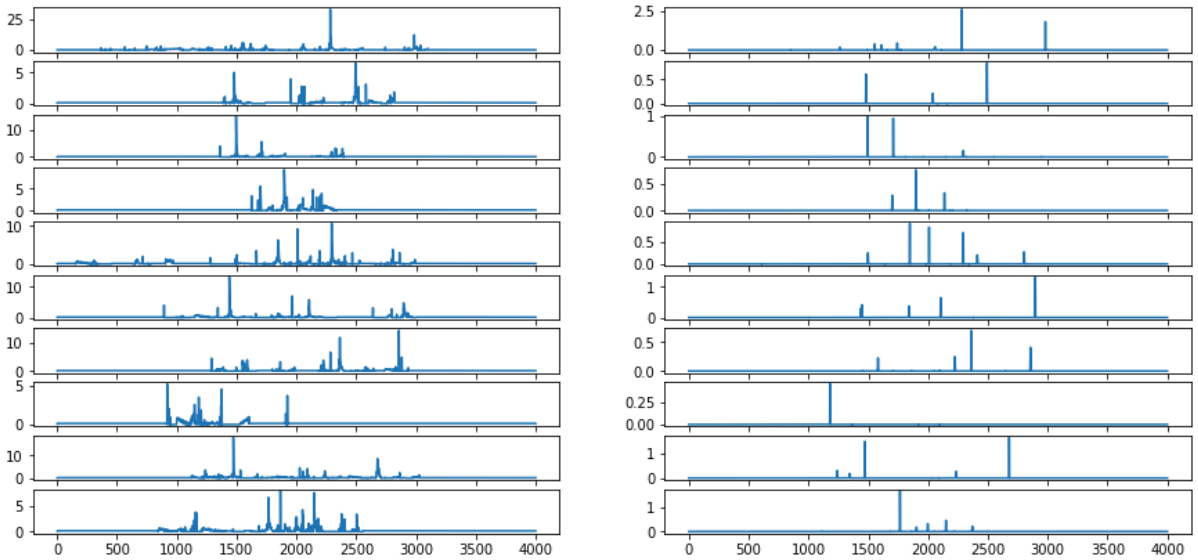


The loss and false positivity rate was going down with this change however the efficiency was also going down. On plotting the output I realised the data was just overfitting and generating just a flat line

HYPER PARAMETER TUNING - 2 - Reducing the learning rate to 1e-06

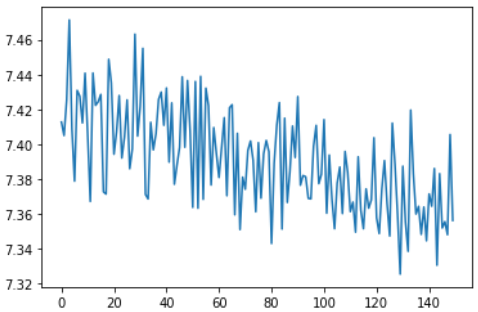


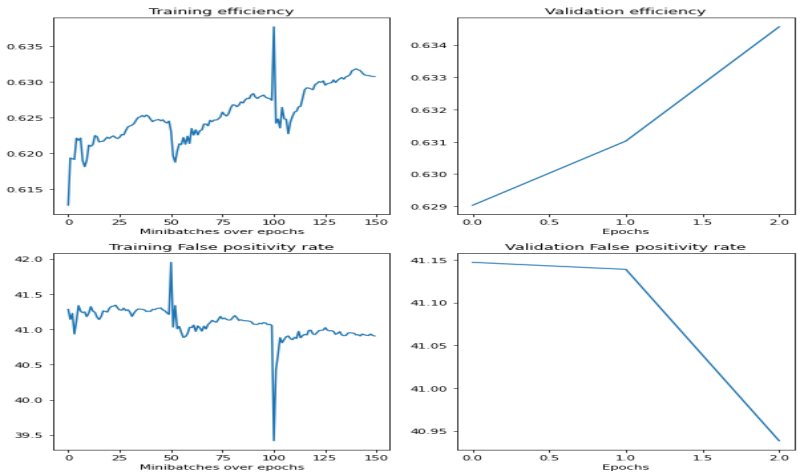


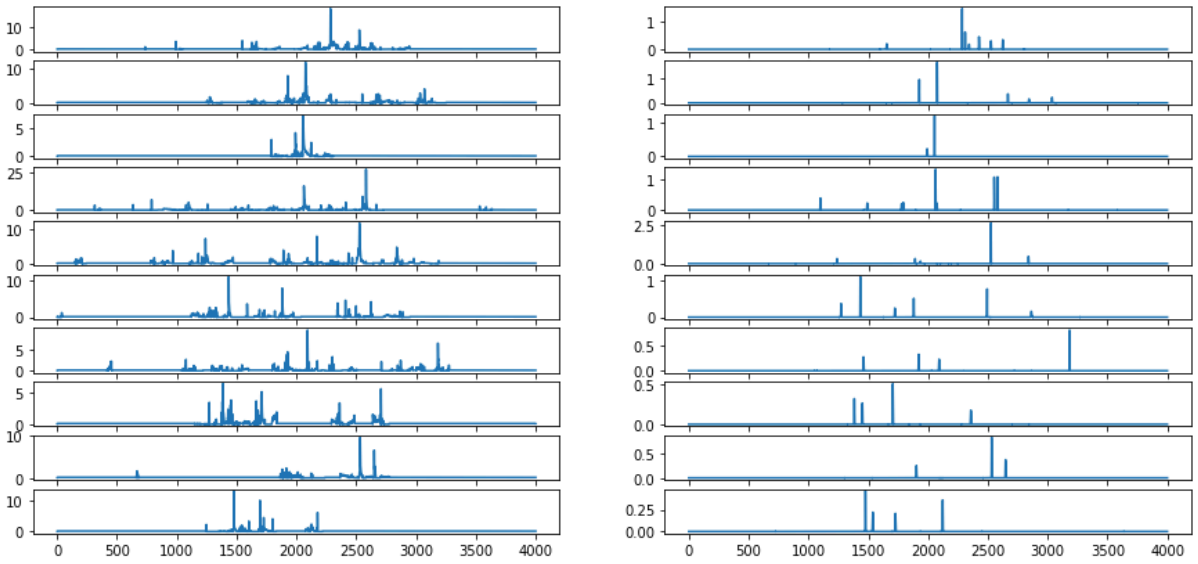


This was a little better than the previous tweak however there are way more excitations than that are required which implies there are far more false positive excitations

HYPERPARAMETER TUNING - 3 - Increase Epochs







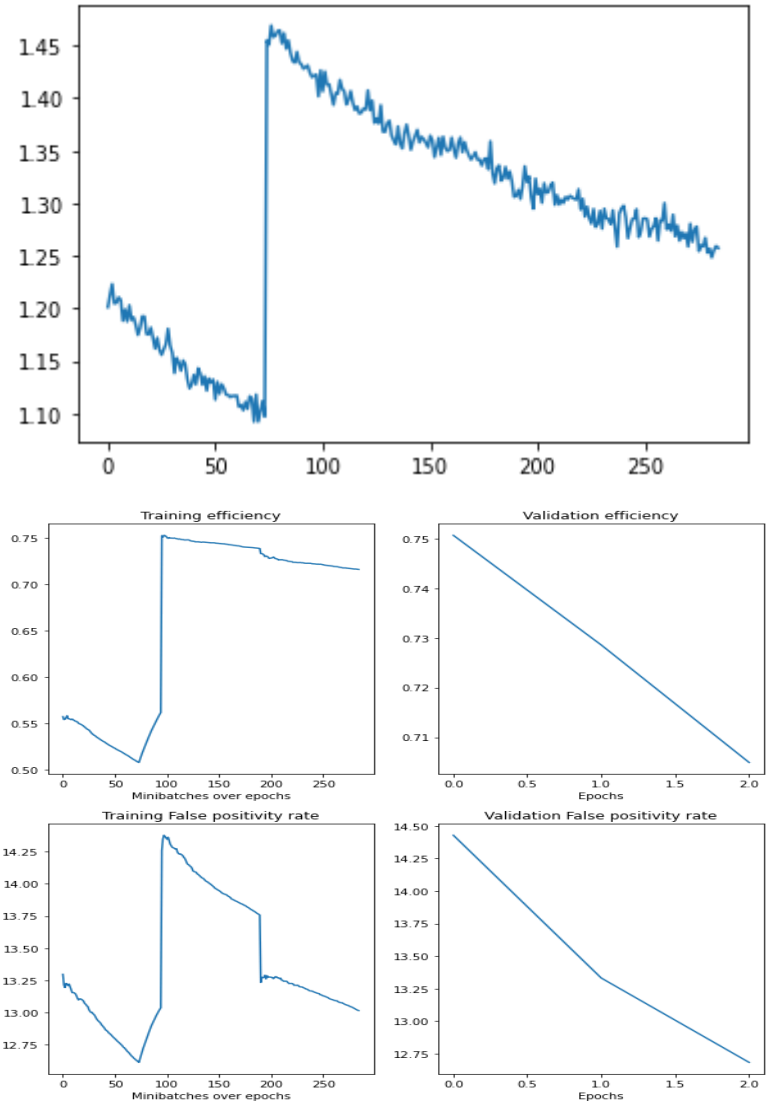
On first look, this would look to be performing well as the efficiency was increasing and fp rate is going down but on closer inspection, we can see that the false positive rate is 41 so, the model is getting exited many more times than is needed.

HYPERPARAMTER TUNING - 4

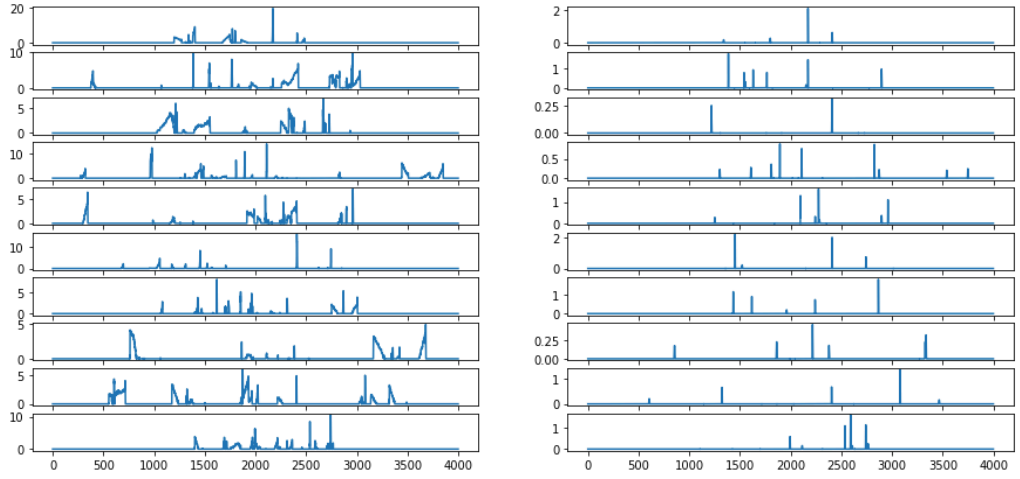
After this point, I tried multiple other things, not plotting graphs here as the performance didn't improve because of them but they are present in the notebook. The things I tried were

1. Increase the number of residual blocks
2. Train with only one residual block
3. Increase Channels
4. Decrease Channels

Impact of a small dataset

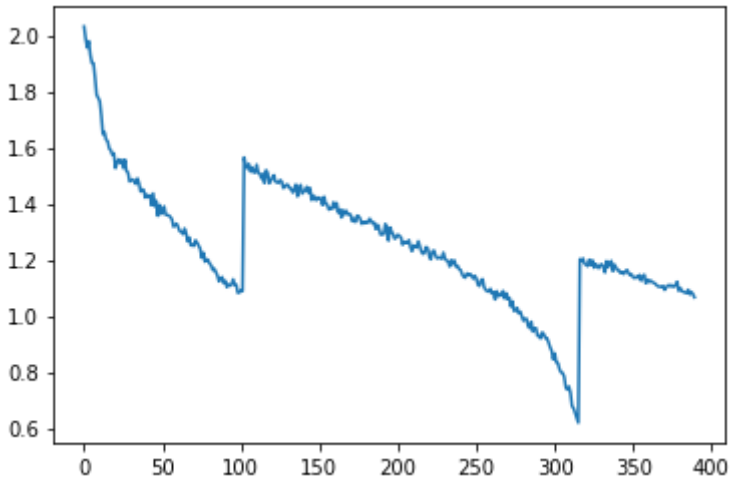


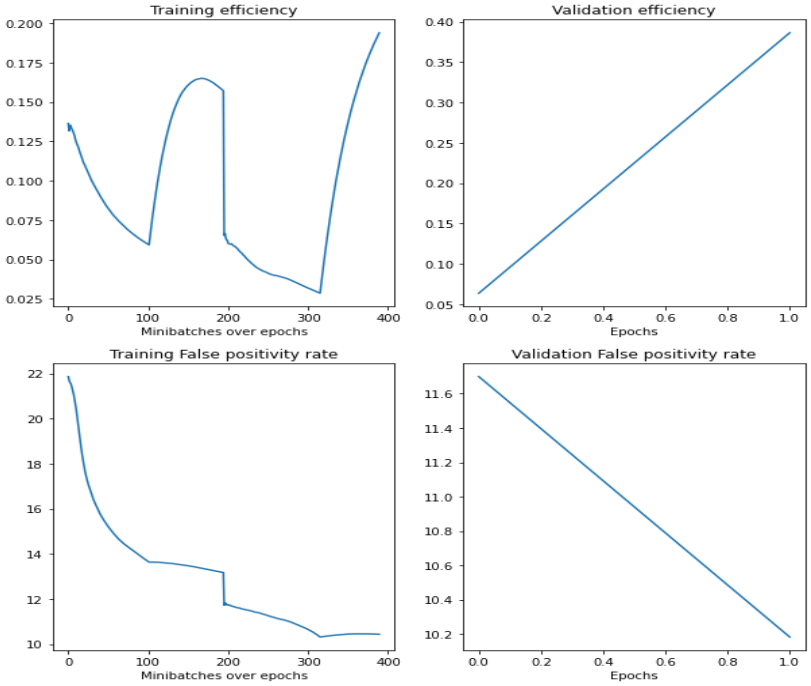
Generated Output



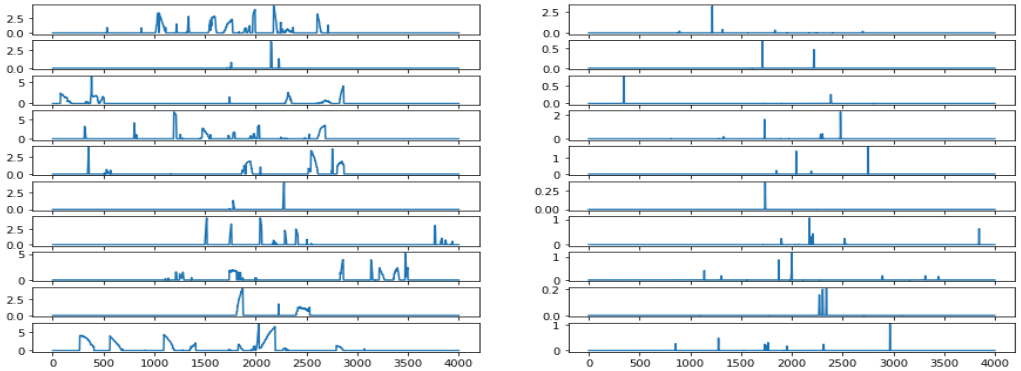
The small dataset got a very good efficiency of 75% but it created way too many false positive excitations as well. The loss was also decent at 1.30

Impact of the medium dataset



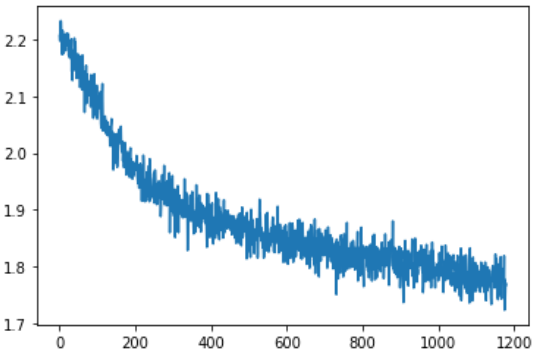


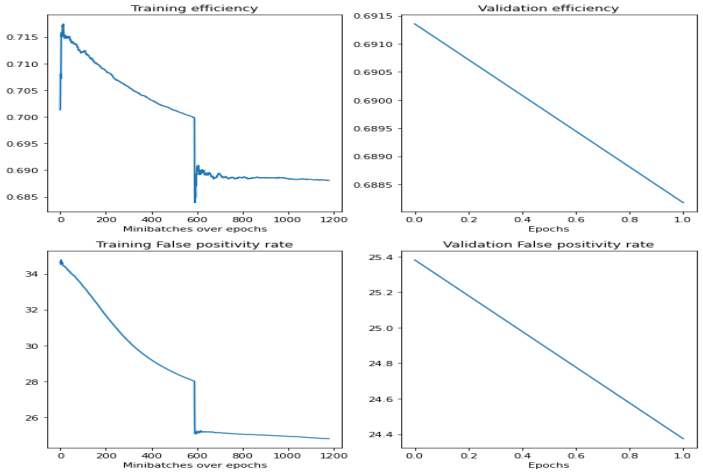
Generated Output



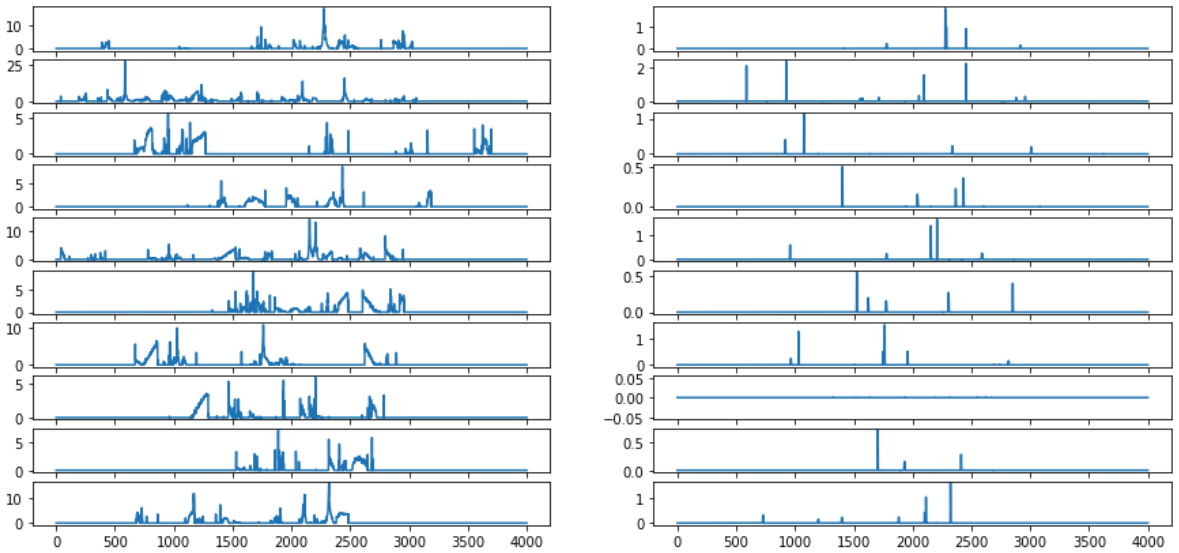
This model also would look like it is doing pretty good at the first look on a medium-sized dataset but on closer look, there are a lot more false positives than efficiency, though the efficiency is good it is getting excited way more times than needed.

Impact of Large Dataset





Generated Output



Even the large dataset was generating more false positive excitations than that are required.

Conclusion

Even though ResNet Reducer had better efficiency, I conclude that the first model Convoluted Encoder was much better because it has far fewer false positives and a better loss.